ParaSoft®
jtest!™

jtest!
User's Guide

ParaSoft® jtest!™

Table of Contents

Introduction

Introduction 1
jtest! Installation & Licensing 2
Contacting ParaSoft 3
Quick Start Guide 4

Testing with jtest!

Testing A Single Class 6
Testing A Class - A Simple Example 8
Understanding the Errors Found Panel 13
Understanding and Customizing Class Test Results 15
Testing a Set of Classes 17
Testing A Set of Classes - Example 20
Understanding the Results Panel 22
Understanding and Customizing Project Test Results 26
Opening One of A Project's Classes in the Class Testing UI 28
Editing the Class Test Parameters From the Project Testing UI 29

Static Analysis

About Static Analysis 30
Performing Static Analysis 31
Customizing Static Analysis 32

Dynamic Analysis

About Dynamic Analysis 34
Performing Dynamic Analysis 35
Customizing Dynamic Analysis 36
Setting an Object to a Certain State 37

White-Box Testing

About White-Box Testing 38
Performing White-Box Testing 39

Black-Box Testing
About Black-Box Testing 40
Performing Black-Box Testing (From Class Testing UI) 41
Performing Black-Box Testing (From Project Testing UI) 43
Adding Inputs 45
Specifying Imports 48

Regression Testing
About Regression Testing 49
Performing Regression Testing 50

Miscellaneous Topics

Saving and Restoring Tests Parameters 51
Viewing Test History 52
Viewing Context-Sensitive Help 53
Viewing, Editing, or Compiling A Source 54
Viewing and Validating Test Cases 55
Viewing A Report of Results 60
Viewing the Contents of A Results Directory 62
Running jtest! in Batch Mode 64

Customizing jtest!

Customizing Test Parameters 65
Customizing Reporting of Violations 66
Static Analysis Suppressions 67
Dynamic Analysis Suppressions 69
Customizing System Settings 72

jtest! UI Help

jtest! UI Overview 73
Trees 74
Cursors 75

Class Testing UI

Class Testing UI 76
Menu Bar 77
Toolbar 79
Class Name Panel 82
Test Progress Panel 83
Errors Found Panel 85

Project Testing UI

Project Testing UI 86
Project Testing UI Menu Bar 87
Project Testing UI Toolbar 89
Project Testing UI Controls Panel 93
Project Testing UI Results Panel 94

Test Parameter Windows

Global Test Parameters 95
Global Test Parameters - Static Analysis 96
Global Test Parameters - Dynamic Analysis 99
Global Test Parameters - Common Parameters 104
Class Test Parameters 108
Class Test Parameters - Static Analysis 109
Class Test Parameters - Dynamic Analysis 110
Class Test Parameters - Common Parameters 117
Project Test Parameters 119
Project Test Parameters - Static Analysis Parameters 120
Project Test Parameters - Dynamic Analysis Parameters 121
Project Test Parameters - Common Parameters, Search Parameters, Classes in Project 124

Tools

Find Classes UI 128

Reference

Built-in Static Analysis Rules 130

Tutorial

jtest! Tutorial 186
Lesson 1 - Performing White-Box Testing 187
Lesson 2 - Performing Static Analysis 193
Lesson 3 - Performing Black-Box Testing 197
Lesson 4 - Performing Regression Testing 209
Lesson 5 - Testing a Set of Classes 212

Introduction

jtest! is a comprehensive class testing tool that makes it easy to perform every type of testing that can be performed on a class.

jtest! performs the following types of tests:

- Static Analysis
- White-Box Testing
- Black-Box Testing
- Regression Testing

jtest! can perform all of these tests on either a single class or a set of classes with a single click of button. Static analysis, white-box testing, and regression testing are completely automatic; they require no user intervention at all. Black-box testing with jtest! is significantly easier (and more effective) than with any other tool. jtest! automatically generates a core set of inputs, strategically designed to achieve as full coverage as possible; you can augment these inputs with your own inputs. jtest! then automatically executes all the inputs and displays the outcomes for those inputs in a simple tree representation. You can then view the outcomes and validate them with one click of a button, so when jtest! performs subsequent tests on this class, it will be able to notify you when specification and regression testing errors occur.

With jtest!, you can not only detect errors, but also prevent errors and ensure that errors do not enter code when it is modified-- all automatically. Static analysis prevents errors by enforcing Java coding standards that reduce the opportunity for errors to enter the code; it also improves software maintainability and reusability. By testing a class with inputs based on the code's internal structure, white-box testing determines if a class is constructed strongly enough to perform well under all circumstances-- even those that you do not foresee. White-box testing alerts you to crash causing errors as well as hidden weaknesses that may cause problems if not fixed. Black-box testing allows you to test if your class performs according to specification, and regression testing checks that modifications do not introduce errors into your classes.

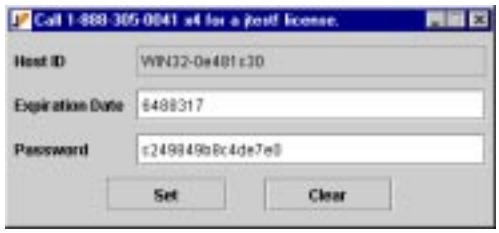
jtest! Installation & Licensing

To install jtest!, simply run the jtest_win32.exe program (this is the file that you downloaded from the ParaSoft Web/FTP site), and follow the onscreen directions. The program will automatically install jtest! on your computer.

A jtest! license must be installed before you can begin using jtest!.

To install a jtest! license on your machine:

1. Launch jtest! by double-clicking the **jtest!** icon on your desktop. The Class Testing UI and the License window will open.



2. Call **1-888-305-0041 x4** to get your license.
3. In the License window, enter the expiration date and password.
4. Click **Set** to set and save your license.

Contacting *ParaSoft*

ParaSoft is committed to providing you with the best possible product support for jtest!. If you have any trouble installing or using jtest!, contact us at:

USA Headquarters
Tel: (888) 305-0041
Fax: (626) 305-9048
Email: quality@parasoft.com
Web Site: http: //www.parasoft.com

Para*Soft* France
Tel: +33 (0) 1 60 79 51 51
Fax: +33 (0) 1 60 79 51 50
Email: quality@parasoft-fr.com

Para*Soft* Germany
Tel: +49 (0) 78 05 95 69 60
Fax: +49 (0) 78 05 95 69 19
Email: quality@parasoft-de.com

Para*Soft* UK
Tel: +44 171 288 66 00
Fax: +44 171 288 66 02
Email: quality@parasoft-uk.com

Quick Start Guide

jtest! fully automates white-box testing, static analysis, and regression testing. The only user intervention required to perform these tests on a class or set of classes is telling jtest! what class or classes to test, clicking the **Start** button, and looking at the test results. When you perform only these fully automated types of tests, the amount of user intervention required is independent of the number of classes tested.

Because it is impossible to fully automate black-box testing, some user intervention is required if you want to run black-box tests along with the above tests. Specifically, you may want to add your own inputs, and you must indicate the correct input/output relation for each class. This means that the amount of user intervention required is proportional to the number of classes on which you want to perform black-box testing.

To test your class(es) using fully automatic white-box testing, static analysis, and regression testing, perform the following steps:

1. Open the appropriate UI for your test.
    - To test a single class, open the Class Testing UI by launching jtest!
    - To test a set of classes, open the Project Testing UI by launching jtest!, then clicking the **Project** button.
2. Indicate what class or set of classes you want to test.
3. Save your test parameters by choosing **Save** from the **File** menu.
4. Test the class for the first time by clicking the **Start** button.
   The first time you test a class, jtest! will...
    - Perform static analysis.
    - Perform white-box testing.
    - Automatically generate a set of inputs (e.g., a test suite), and store the outcomes for those inputs.
5. Review the class test results or project test results, then...
    - Correct errors found, or suppress reporting of errors you do not want reported in future test runs.
    - (optional) Customize test parameters.
6. Rerun the test when a class is modified. To do this...
    - Choose **File> Open** in the UI that you used for the original test.
    - Click **Start**.
      When the test is run this time, and all additional times, jtest! will...
        - Perform static analysis.
        - Perform white-box testing.

- Perform regression testing by comparing the current test case outcomes with those obtained during the initial test run.

If you also want to perform black-box testing, perform the following additional steps:

7. Add user-defined test cases.
   Adding user-defined test cases involves examining all the classes under test and performing the following tasks for each class:
   - View the automatic inputs generated by jtest! by opening the View Test Cases window.
   - Design additional test cases.
   - Enter additional test cases in jtest!'s Class Test Parameters window.
8. Rerun the test by clicking the **Start** button.
   When the test is run, jtest! will perform all the tests it performed in previous test runs, plus it will perform black-box testing for the user-defined test cases and determine the output for the user-defined test cases.
9. Specify the correct outcomes for all user-defined and (optional) automatically generated test cases.
   Specifying correct outcomes involves performing the following tasks for each class and test case:
   - View the test case input and outcomes in the View Test Cases window.
   - Validate correct outcomes or set the correct value for incorrect outcomes by right-clicking the appropriate outcome node and selecting the appropriate command from the shortcut menu.
   - (optional) Specify additional inputs to check.
10. When the class is modified, rerun the test by restoring test parameters and clicking the **Start** button.
    When you rerun the test, jtest! will check for specification and regression testing errors by comparing validated outcomes with their specified values, and nonvalidated outcomes with their previous values. jtest! will also continue to test for uncaught runtime exceptions and static analysis errors.

Testing A Single Class

Two fundamental steps are required to run any test:

- Indicating what class to test.
- Starting the test.

In addition to performing these two steps, you may choose to customize test parameters that alter what types of tests are performed and how tests are performed.

For details on specific types of tests, see the following topics:

- About Static Analysis
- About Dynamic Analysis
    - About White-Box Testing
    - About Black-Box Testing
    - About Regression Testing

Indicating what class to test

There are several easy ways to indicate what class to test:

- Browse for the class by...
    1. Clicking the **Browse** button in the Class Testing UI's Class Name panel.
    2. Locating and selecting the file you want to test in the file viewer.
    3. Clicking **Open**.
- Enter the class directly in the Class Testing UI's Class Name panel by entering the fully qualified name of the class to test (without the .class extension) in the **Class Name** field.
  Note: It is recommended that the class to be tested is selected using the **Browse** button. When a class is selected using the **Browse** button, the working directory is set to the root directory of the class's package.
- Use the Find Classes UI to find available classes, then double-click the name of the appropriate test in the lower panel of the Find Class UI. This will set up a test for the selected class in the Class Testing UI.

Starting the test

To run a test, click the **Start** button in the Class Testing UI toolbar.

By default, jtest! automatically performs all steps required for static analysis, white-box testing, and regression testing (on all test runs after the first). jtest! automatically tests for specification errors as long as test case outcomes have been validated.

If you only want to run Static Analysis, right-click the **Start** button and choose **Static Analysis Only** from the shortcut menu. This will cause jtest! to only perform static analysis (even if parameters are set to perform both static and dynamic testing).

If you only want to run Dynamic Analysis, right-click the **Start** button and choose **Dynamic Analysis Only** from the shortcut menu. This will cause jtest! to only perform dynamic analysis (even if parameters are set to perform both static and dynamic testing).

Results

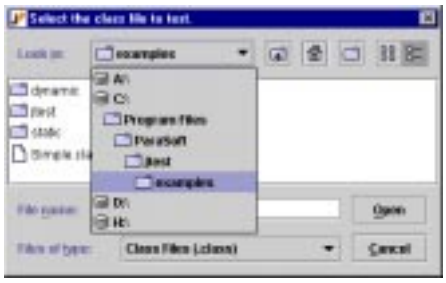
Results are displayed in the Errors Found Panel. To learn more about this panel's branches and available options, see Understanding the Errors Found Panel and Understanding and Customizing Class Test Results.

Testing A Class - A Simple Example

The following example demonstrates a fully automatic way to test a single class.

Static and Dynamic Analysis

1. Open jtest!
2. Browse to Simple.class in the jtest! examples directory using the **Browse** button in the Class Name panel.

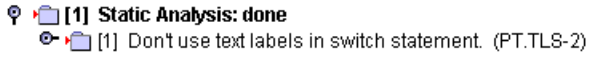


3. Click the **Start** button in the toolbar.

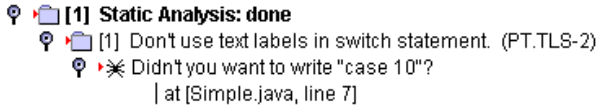
jtest! will perform static analysis and white-box testing on the class. A dialog box will open to notify you when testing is complete. Information on test progress will be displayed in the Test Progress panel. Errors found will be reported in the Errors Found panel.

Once testing is completed, you will see two errors reported in the Errors Found panel:

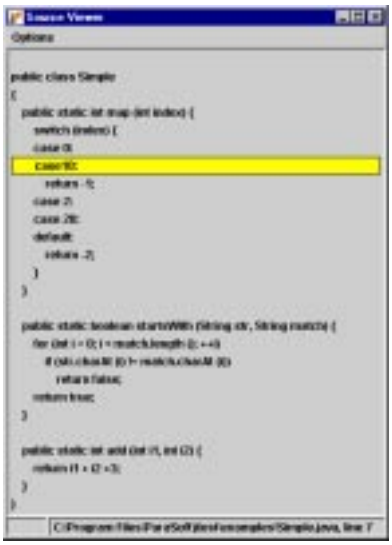
1. **Static Analysis:** The Errors Found panel will display the following error:



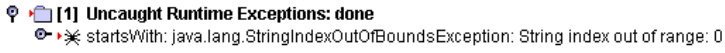
Expanding all of this violation message's branches reveals more information about the violation found. This error reveals that the developer inadvertently wrote "case10" instead of "case 10". If the class is not fixed, it will give incorrect results when it is passed the value 10.


[1] Static Analysis: done
[1] Don't use text labels in switch statement. (PT.TLS-2)
Didn't you want to write "case 10"?
at [Simple.java, line 7]

To view the source code of the class with the line containing the error highlighted, double-click the node containing the error's file/line information.



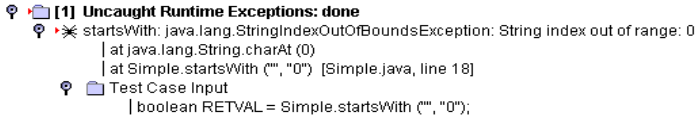
2. **Uncaught Runtime Exceptions:** The Errors Found panel will list the following error:


[1] Uncaught Runtime Exceptions: done
startsWith: java.lang.StringIndexOutOfBoundsException: String index out of range: 0

This error message reveals that there is some input for which the class will throw an uncaught runtime exception at run-time. This could cause the application running this class to crash.

To see a stack trace like the one the Java virtual machine gives when an uncaught runtime exception is thrown, expand this branch.

To see an example usage of this class that leads to the reported uncaught runtime exception, expand the **Test Case Input** branch.


[1] Uncaught Runtime Exceptions: done
startsWith: java.lang.StringIndexOutOfBoundsException: String index out of range: 0
| at java.lang.String.charAt (0)
| at Simple.startsWith ("", "0") [Simple.java, line 18]
Test Case Input
| boolean RETVAL = Simple.startsWith ("", "0");

The error shows us that the "startsWith" method is implemented incorrectly. The method should return false for the argument "" and "0" instead of throwing a runtime exception.

If the error is not fixed, any application using this class will eventually crash or give incorrect results when used.

To view the source code of the class with the problematic line of the stack trace highlighted, double-click the node containing the exception's file/line information.

Regression Testing

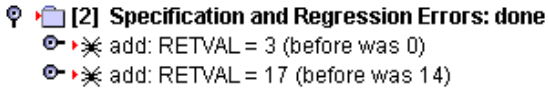
jtest! doesn't display any regression errors on the first run through a class because it is impossible for jtest! to give a regression error the first time it runs a class. Regression testing checks that the class outcomes don't change, so it always needs a first run for reference.

To see how regression testing works, we will introduce an error into Simple.java and run it again.

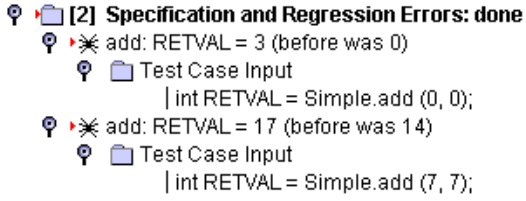
1. Copy Simple_re.java into Simple.java (both classes are located in the jtest! examples directory).

2. Recompile Simple.java.
3. Retest Simple.java.

Now, along with the other errors, jtest! reports the following regression errors in the Errors Found panel:



Expand the error messages to see the inputs for which these regression errors occur. The first error tells us that the code has changed and that the method "add" is now returning 3 instead of 0 for the input 0, 0. The second error reveals that the method "add" is now returning 17 instead of 14 for the input 7,7.


[2] Specification and Regression Errors: done
add: RETVAL = 3 (before was 0)
Test Case Input
int RETVAL = Simple.add (0, 0);
add: RETVAL = 17 (before was 14)
Test Case Input
int RETVAL = Simple.add (7, 7);

Understanding the Errors Found Panel

All class test results are displayed in the Errors Found panel of the Class Testing UI. The contents of this panel are described below, and in context-sensitive help. If commands are available by right-clicking a particular node, a right-click icon will open when your cursor is placed over that node.

The Class Testing UI also provides you with a variety of ways to gain additional information about the test results and to customize what results are reported the next time this test is run. For more information about these options, see Understanding and Customizing Class Test Results.

The tree displayed in this panel contains the following information:

- **Static Analysis:** Displays the number of violations found while performing static analysis.
    - Number of violations found, grouped by rule violated (Rule ID is displayed in parentheses).
        - jtest! rule violation message. To suppress this message or view the associated rule description, right-click this node then choose the appropriate command from the shortcut menu.
        - File/line number where violation occurred. To view or edit the source code, right-click this node then choose the appropriate command from the shortcut menu.

- **Uncaught Runtime Exceptions:** Displays the number of uncaught runtime exceptions that jtest! found while performing dynamic analysis. Each uncaught runtime exception is followed by a full stack trace, as well as an example input leading to this exception.
    - Specific exceptions found.
        - Stack trace information. To view or edit the source code, right-click this node then choose the appropriate command from the shortcut menu. (If the file and line number information is missing, recompile the class with debug information).
        - Input that defines the test case. (For automatic test cases, this is the calling sequence; for user defined test cases, this is the input for each argument).
- **Specification and Regression Errors:** Displays specification and regression errors found by jtest! while performing dynamic analysis. The errors are calculated by comparing the test case outcomes of this run with those of previous runs or those specified by the user. (To view the reference outcomes, open the Class Test Parameters window and browse to **Dynamic Analysis> Test Case Evaluation> Specification and Regression Test Cases**).
    - Specific specification and regression errors found.
        - Input that defines the test case. (For automatic test cases, this is the calling sequence; for user defined test cases, this is the input for each argument).

Note: All error messages are marked with a bug icon.

Understanding and Customizing Class Test Results

The Class Testing UI provides you with a variety of ways to learn more about the test results and to customize what results are reported the next time this test is run. Several actions that you might want to perform when viewing results include:

- **View/Evaluate test cases:** View and/or evaluate the automatically-generated and user-defined test cases used to test this class by clicking the **View Test Cases** button in the toolbar.
- **View the source responsible for a rule violation or exception:** View the source of the error, with the problematic line highlighted, by double-clicking the file/line information for the error in the Errors Found panel.
- **View a description of a violated rule:** View a description of a violated static analysis rule, along with an example of that rule and a suggested repair, by right-clicking a static analysis error message, and choosing **View Rule Description** from the shortcut menu that opens.
- **View the stack trace of an uncaught runtime exception:** View a stack trace like the one the Java virtual machine gives when an uncaught runtime exception is thrown by opening the node containing the uncaught runtime exception.
- **View the calling sequence of an uncaught runtime exception:** View an example usage of the class that leads to the reported uncaught runtime exception by opening the **Test Case Input** node for any uncaught runtime exception.
- **View the input for each argument that resulted in an error:** View the input for each argument in user-defined tests cases by opening the **Test Case Input** node for any specification or regression testing error or uncaught runtime exception error.
- **View an example source:** View a full test case which can be compiled and, in most circumstances, executed to see the actual exception by right-clicking the **Test Case Input** node and choosing **View Example Source** from the shortcut menu.
- **View a report:** View a report file by clicking the **Report** button in the Class Testing UI toolbar.
- **Gauge coverage:** There are two ways to gauge a test's coverage:
    1. Review the coverage data displayed in the Test Progress panel.
    2. Display and review the report file.
       The report file contains, among other information, the annotated source code for the tested class. This may be used to determine what lines jtest! tested and what lines it did not test.

- **Modify test case evaluation:** If a reported uncaught runtime exception is actually the correct behavior of the class, or if you want jtest! to ignore the outcome of an input while checking for specification and regression errors, right-click the error message, then choose the appropriate command from the from the shortcut menu.
    - To indicate that a reported error is not an error, choose **Not an Error**.
    - To tell jtest! to ignore the outcome for this input, choose **Ignore this Outcome**. If you choose this option, the outcome will not be used for comparisons when searching for specification or regression errors. Also, no uncaught runtime exceptions will be reported for this test case input.
    - To have jtest! ignore all outcomes in a class's **Uncaught Runtime exceptions** or **Specification and Regression Errors** node, or to indicate that all errors contained in a class's **Uncaught Runtime Exceptions** or **Specification and Regression Errors** node are not actual errors, right-click the appropriate node and choose **Set All to: Not an Error** or **Set All to:Ignore this Outcome**.
- **Suppress messages:** You can also suppress the reporting of a single specific exception or static analysis violation by right-clicking the error message that you do not want reported in future test runs and choosing **Suppress** from the shortcut menu. This automatically adds the suppression to the appropriate Suppressions Table.

Testing a Set of Classes

In jtest!'s Project Testing UI, you can automatically test all the classes contained in any directory, jar file, or zip file with a single click. jtest! automatically searches the specified directory, jar file, or zip file, and tests all classes that it finds.

It is possible to perform all testing-related activities in the Project UI. The Project Testing UI contains all results for all classes tested, and allows you to access the same features that are available in the Class Testing UI.

If you want to test an entire project then focus on results found on a class-by-class basis, you can test the project in the Project Testing UI, then open the class(es) you want to focus on in the Class Testing UI.

By default, jtest! will not test a class it has previously tested unless that class has been modified since the last test. jtest! determines whether or not a class has changed by checking that both the .class file and the .java file contents have not changed. Timestamps are not considered. To force jtest! to test all classes on every test, disable the **SkipClassesAlreadyTested** node in the **Search Parameters** branch of the Project Test Parameters tree.

Two fundamental steps are required to test a set of classes:

- Indicating where jtest! should start finding and testing classes.
- Starting the test.

In addition to performing these two steps, you may choose to customize test parameters that alter what types of tests are performed and how tests are performed.

For details on specific types of tests, see the following topics:

- About Static Analysis
- About Dynamic Analysis
    - About White-Box Testing
    - About Black-Box Testing
    - About Regression Testing

Indicating where jtest! should start

1. Open the Project Testing UI by clicking the **Project** button in the Class Testing UI toolbar, or by choosing **Project Testing UI** from the **File** menu in the Class Testing UI menu bar.

2. In the **Search In** field of the Project Testing UI, enter the name of the directory, zip file, or .jar file where you want jtest! to start searching for classes to test. To browse for the directory, jar file, or zip file that you want jtest! to start searching and testing, click the **Browse** button, locate and select the desired directory, jar file, or zip file in the file viewer, then click **Open**.
   If the parameter is a directory, jtest! will recursively traverse the path's subdirectories, zip files, and jar files, searching for and testing any classes it finds.
   If the parameter is a jar or zip file, jtest! will open the file and search it for classes in which to find errors.
3. (OPTIONAL) If you want to restrict the classes that jtest! tests...
   - Use the **Filter-in** field to tell jtest! to only find and test classes that match the given expression. Use the * (star) character to match zero or more characters.
     For example, if you want jtest! to only look for classes in the util package, enter the following parameter in this field
     `util.*`
     When this field is left empty, all classes found will be tested.

     OR

   - Use the Skip List to indicate specific classes that you want jtest! to skip. The Skip List is accessible by clicking the **Skip List** node in the **Search Parameters** branch of the Project Test Parameters panel. Clicking this node opens a dialog box that lets you enter the names of specific classes in your project that you do not want tested.

     OR

   - Use Test Only List to indicate specific classes that you want jtest! to test. The Test Only List is accessible by clicking the **Test Only List** node in the **Search Parameters** branch of the Project Test Parameters panel. Clicking this node opens a dialog box that lets you enter the names of specific classes in your project that you want tested.

Starting the test

Testing a project is simple. To prompt jtest! to start finding and testing classes, simply click the **Start** button.

By default, jtest! automatically performs all steps required for static analysis, white-box testing, and regression testing (on all test runs after the first). jtest! automatically tests for specification errors as long as test case outcomes have been validated.

If you only want to run Static Analysis, right-click the **Start** button and choose **Static Analysis Only** from the shortcut menu. This will cause jtest! to only perform static analysis (even if parameters are set to perform both static and dynamic testing).

If you only want to run Dynamic Analysis, right-click the **Start** button and choose **Dynamic Analysis Only** from the shortcut menu. This will cause jtest! to only perform dynamic analysis (even if parameters are set to perform both static and dynamic testing).

To have jtest! stop finding and testing classes, click the **Stop** button.

To have jtest! temporarily pause (or resume) finding and testing classes, click the **Pause** button. If you pause testing, jtest! will finish testing the current class before pausing.

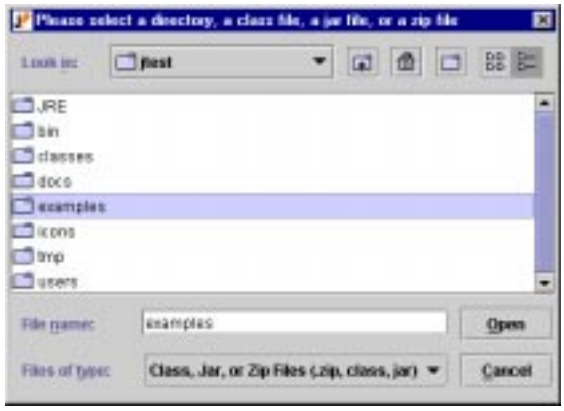
Results

Results are displayed in the Project Testing UI's Results panel. To learn more about this panel's branches and available options, see Understanding the Results Panel and Understanding and Customizing Project Test Results.

Testing A Set of Classes - Example

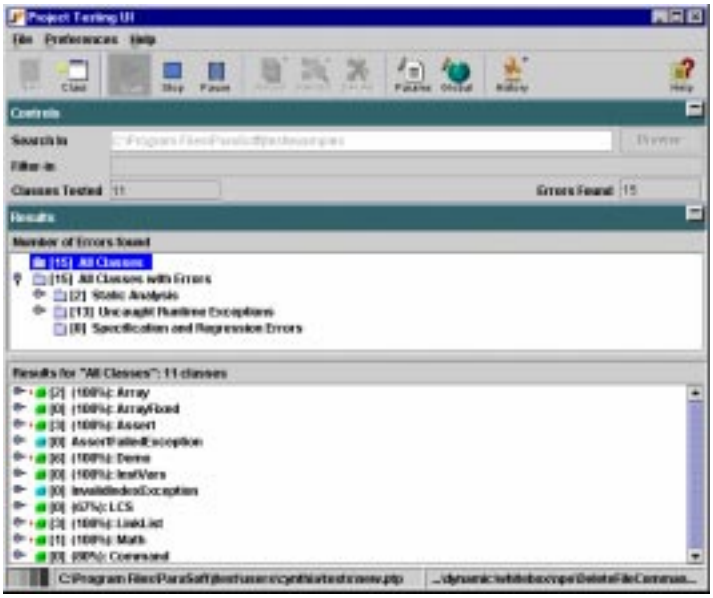
The following example demonstrates the completely automatic mode of testing a set of classes.

1. Open jtest!
2. Click the **Project** button in the Class Testing UI toolbar. The Project Testing UI will open.
3. Click the **Browse** button in the Project Controls panel, locate the jtest! examples directory in the file viewer that opens, then click **Open**



4. Click the **Start** button in the Project UI toolbar.

jtest! will then start finding and testing classes.



Results are displayed in the Project Testing UI's Results panel. To learn more about this panel's branches and available options, see Understanding the Results Panel and Understanding and Customizing Project Test Results.

Understanding the Results Panel

All results from a project test are displayed in the Results panel of the Project Testing UI. The contents of this panel are described below, and in context-sensitive help. If commands are available by right-clicking a particular node, a right-click icon will open when your cursor is placed over that node.

The Project Testing UI also provides you with a variety of ways to gain additional information about the tests performed and test results and to customize what results are reported the next time this test is run. For more information about these options, see Understanding and Customizing Project Test Results.

Number of Errors Found Window

This window displays the distribution of the errors found. The tree in this window contains a node for every type of error that jtest! detects, along with the number of that type of error found in the project test.

This window allows you to determine what results are displayed in the Results For All Classes window. To restrict results displayed in the lower results window to those of classes that contain a certain type of result (such as All Classes With Errors, Uncaught Runtime Exceptions, or java.lang.NullPointer Exceptions) perform the following steps:

1. Right-click the node that describes the type of results you want to view. A shortcut menu will open.
   Note: If the type of result you want to view is not visible, expand the All Classes With Errors tree branches.
2. Choose **Show Results for this category** from the shortcut menu.

The node that describes the types of results displayed in the lower Results window will be highlighted in blue.

Results For All Classes Window

This window lists the results for the current project test. The results are organized into a tree. Each tree leaf corresponds to the results for one class. Public classes have a green block to the left of their name; package private classes have a blue block to the left of their name. To view results for a class, expand the branches that correspond to that class.

Each class has two main sub-branches:

- **Test Progress:** Contains test status and coverage information.
- **Errors Found:** Contains information about errors found.

Test Progress

The Test Progress branch contains the following information:

- **Static Analysis:** Displays the progress of Static Analysis tests.
    - **Number of Rules Analyzed:** Displays number of static analysis rules analyzed.
- **Dynamic Analysis:** Displays the progress of Dynamic Analysis tests.
    - **Number of Test Cases Executed:** Displays the total number of test cases executed.
        - **Automatic:** Displays the number of automatically generated test cases executed.
        - **User Defined:** Displays number of user defined test cases executed.
    - **Number of outcome comparisons:** Number of class outcomes compared during black-box and regression testing.
    - **Total Coverage:** Displays the cumulative coverage jtest! achieved. jtest! performs data coverage for the generated input categories; this means that the parts of the class that have been covered are thoroughly tested with respect to those inputs. The coverage reported is relative to the classes that have been accessed for the paths jtest! has tried. If some part of the class is not covered, it means that jtest! has not yet found a path leading to those statements or no path leads to those statements. In class testing mode, approximately 50% of a class's code can be tested by jtest! Sometimes jtest! will be able to test 100% of the class, and sometimes it will test less than 50% of the class.
    - **Multi-condition branch:** Displays coverage achieved on multi-condition branches.
        - **Method:** Displays coverage achieved on methods.
        - **Constructor:** Displays coverage achieved on constructors.

      Note: Dynamic coverage is shown only for classes that dynamic analysis has been performed on (by default, dynamic analysis is only performed on the public classes. Static analysis is performed on all classes found (public and non-public).

Errors Found

The Errors Found branch is organized like the Errors Found tree in the Class Testing UI Errors Found panel. It contains the following information:

- **Static Analysis:** Displays the number of violations found while performing static analysis.
    - Number of violations found, grouped by rule violated (Rule ID is displayed in parentheses).
        - jtest! rule violation message.
        - File/line number where violation occurred.

- **Uncaught Runtime Exceptions:** Displays the number of uncaught runtime exceptions that jtest! found while performing dynamic analysis. Each uncaught runtime exception is followed by a full stack trace, as well as an example input leading to this exception.
    - Specific exceptions found.
        - Stack trace information. (If the file and line number information is missing, recompile the class with debug information).
        - Input that defines the test case. (For automatic test cases, this is the calling sequence; for user defined test cases, this is the input for each argument).
- **Specification and Regression Errors:** Displays specification and regression errors found by jtest! while performing dynamic analysis. The errors are calculated by comparing the test case outcomes of this run with those of previous runs or those specified by the user. (To view the reference outcomes, open the Class Test Parameters window and browse to **Dynamic Analysis> Test Case Evaluation>Specification and Regression Test Cases**).
    - Specific specification and regression errors found.
        - Input that defines the test case. (For automatic test cases, this is the calling sequence; for user defined test cases, this is the input for each argument).

Note: All error messages are marked with a bug icon.

Understanding and Customizing Project Test Results

jtest!'s Project Testing UI provides you with a variety of ways to gain additional information about test results, as well as to customize what results are reported the next time this test is run. Most options are accessed via shortcut menus in the lower Results window. Several actions that you might want to perform when viewing results include:

- **View/Evaluate test cases:** View and/or evaluate the automatically-generated and user-defined test cases used to test this class by right-clicking either the **[Class Name]** node, or the **[Class Name]> Test Progress> Dynamic Analysis> Number of Test Cases Executed** node, and choosing **View Test Cases** from the shortcut menu.
- **View the source responsible for a rule violation or exception:** View the source of the error, with the problematic line highlighted, by double-clicking the file/line information for any error contained within the **Errors Found** branch.
- **View a description of a violated rule:** View a description of a violated static analysis rule, along with an example of that rule and a suggested repair, by right-clicking a static analysis error message, and choosing **View Rule Description** from the shortcut menu that opens.
- **View the stack trace of an uncaught runtime exception:** View a stack trace like the one the Java virtual machine gives when an uncaught runtime exception is thrown by opening the node containing the uncaught runtime exception.
- **View the calling sequence of an uncaught runtime exception:** View an example usage of the class that leads to the reported uncaught runtime exception by opening the **Test Case Input** node for any uncaught runtime exception.
- **View the input for each argument that resulted in an error:** View the input for each argument in user-defined tests cases by opening the **Test Case Input** node for any specification or regression testing error or uncaught runtime exception error.
- **View an example source:** View a full test case which can be compiled and, in most circumstances, executes to see the actual exception by right-clicking the **Test Case Input** node and choosing **View Example Source** from the shortcut menu.
- **View a report:** View a report file by clicking the **Report** button in the Project Testing UI toolbar.
- **Gauge coverage:** Review coverage information by opening the **[Class Name]> Test Progress> Total Coverage** node.

- **Modify test case evaluation:** If a reported dynamic analysis error is actually the correct behavior of the class, or if you want jtest! to ignore the outcome of an input while checking for specification and regression errors, right-click the error message, then choose the desired command from the from the shortcut menu.
    - To indicate that a reported error is not an error, choose **Not an Error**.
    - To tell jtest! to ignore the outcome for this input, choose **Ignore this Outcome**. If you choose this option, the outcome will not be used for comparisons when searching for specification or regression errors. Also, no uncaught runtime exceptions will be reported for this test case input.
    - To have jtest! ignore all outcomes in a class's **Uncaught Runtime exceptions** or **Specification and Regression Errors** node, or to indicate that all errors contained in a class's **Uncaught Runtime Exceptions** or **Specification and Regression Errors** node are not actual errors, right-click the appropriate node and choose **Set All to: Not an Error** or **Set All to:Ignore this Outcome**.
- **Suppress messages:** You can also suppress the reporting of a single specific exception or static analysis violation by right-clicking the error message that you do not want reported in future test runs and choosing **Suppress** from the shortcut menu. This automatically adds the suppression to the appropriate Suppressions Table.
- **Remove a class's results from the Results panel and Results folder:** Remove the results of a class from both the Results panel and the Results Folder (which stores project test results) by right-clicking the **[Class Name]** node, then choosing **Delete** from the shortcut menu.
- **Edit Class Test Parameters:** Modify a specific class's Class Test Parameters by right-clicking the **[Class Name]** node, then choosing **Edit Class Test Parameters** from the shortcut menu.
- **Load in Class Testing UI:** To focus on the errors for a single class, view the class in the Class Testing UI by right-clicking the **[Class Name]** node, then choosing **Load in Class Testing UI** from the shortcut menu.

Opening One of A Project's Classes in the Class Testing UI

There are two ways to open one of a project's classes in the Class Testing UI:

Method 1 (Before or After a Project Test)

1. In the Project Testing UI, click **Params** to open the Project Test Parameters. The Project Test Parameters window will open.
2. Open the **Classes in Project** branch.
3. Right-click the node that corresponds to the class you want to open in the Class Testing UI. A shortcut menu will open.
4. Choose **Load Class in Class Testing UI** from the shortcut menu. The class (test parameters) will be loaded in the Class Testing UI.

Method 2 (After a Project Test Including the Class You Want to Open)

1. If the Project Testing UI's Results panel does not contain test results, open results in the Project Testing UI.
2. In the lower Results panel, right-click the node whose name corresponds to the class you want to open in the Class Testing UI. A shortcut menu will open.
3. Choose **Load in Class Testing UI** from the shortcut menu. The class (test parameters) will be loaded in the Class Testing UI.

Method 3

1. In the Class Testing UI, choose **File> Open** and browse to the “ctp” class test parameters file associated with the class you want to open.

Editing the Class Test Parameters From the Project Testing UI

There are two ways to edit Class Test Parameters from the the Project Testing UI:

Method 1 (Before or After a Project Test)

1. In the Project Testing UI, click **Params** to open the Project Test Parameters. The Project Test Parameters window will open.
2. Open the **Classes in Project** branch.
3. Right-click the node that corresponds to the class whose parameters you want to modify. A shortcut menu will open.
4. Choose **Edit Class Test Parameters** from the shortcut menu. The Class Test Parameters window will open.
5. Modify parameters in the Class Test Parameters window.

Method 2 (After a Project Test Including the Class Whose Parameters You Want to Modify)

1. If the Project Testing UI's Results panel does not contain test results, open results in the Project Testing UI.
2. In the lower Results panel, right-click the node whose name corresponds to the class whose parameters you want to modify. A shortcut menu will open.
3. Choose **Edit Class Test Parameters** from the shortcut menu. The Class Test Parameters window will open.
4. Modify parameters in the Class Test Parameters window.

About Static Analysis

Static analysis can be thought of as consisting of both an automatic code review and automatic coding standards enforcement.

Code reviews have been found to be very useful in uncovering errors and logic flaws in code. In a code review, someone other than the original programmer looks for possible errors and problems in code that already compiles.

Adhering to coding standards is an effective error prevention strategy. Coding standards are rules that ensure that code is written in a way that makes it less error-prone. By adhering to these standards, you significantly reduce the opportunities to introduce errors in the code. Adhering to coding standards enforcement also makes code more readable and easier to maintain.

When performing static analysis, jtest! statically analyzes the classes by parsing the .java source and applying a set of rules to it, then alerting you to any rule violations in the class(es) under test.

Related Topics

Performing Static Analysis

Customizing Static Analysis

Static Analysis Tutorial

Performing Static Analysis

By default, jtest! is configured to perform static analysis. Each time you test a class or set of classes, static analysis (as well as all appropriate dynamic analysis tests) will be performed automatically.

Results

If you tested a project, results are displayed in the **Class Name> Errors Found> Static Analysis** branch of the Project UI's Results panel. To learn more about this panel's branches and available options, see Understanding the Results Panel and Understanding and Customizing Project Test Results.

If you tested a single class, results are displayed in the **Static Analysis** branch of the Errors Found panel. To learn more about this panel's branches and available options, see Understanding the Errors Found Panel and Understanding and Customizing Class Test Results.

Related Topics

About Static Analysis

Customizing Static Analysis

Static Analysis Tutorial

Customizing Static Analysis

You can change the rules that jtest! enforces by enabling or disabling specific rules, or entire rule severity categories.

Note: A description of each rule, as well as an example violation and suggested repair, appears in the Rules section of the Reference Guide. You can also see a description of a rule by right-clicking a rule's node in the Global Parameters tree, then choosing **View Rule Description** from the shortcut menu.

Enabling/Disabling Specific Rules

1. View which rules are currently enabled/disabled.
    - Before setting up a test, open the Global Test Parameters window by clicking the **Global** button.
    - In the Global Test Parameters window, open the **Static Analysis** node.
    - Open the **Rules** node.
    - Open the **Built-in Rules** node.
    - Open the node category whose rules you would like to view.
2. Enable/disable rules by right-clicking the appropriate rule and choosing a command (either **Enable** or **Disable**) from the shortcut menu.

Enabling/Disabling Rule Severity Categories

Each rule has a severity category associated with it (This severity is indicated by the final character in each rule code. For example, UC_AUV-2 has a severity level of 2, and INIT_EIV-5 has a severity level of 5.) There are five severity levels; level 1 is the most severe and level 5 is the least severe. Only severity levels 1, 2, and 3 are enabled by default.

jtest! looks for violations to a rule only if the rule is enabled *and* its severity level is also enabled. Thus, you may also want to configure which severity categories are enabled and disabled. To change this in the Global Test Parameters, open the **Severity Level** node (beneath **Static Analysis> Rules**), and enable/disable categories by right-clicking a category whose status you want to change and choosing **Set to true** or **Set to false** from the shortcut menu.

If you want to customize rules by project, you can enable/disable rules by severity category differently for each class or project test. To do this, set the **Severity** flags in Class Test Parameters - Static Analysis (if you testing a single class) or Project Test Parameters - Static Analysis Parameters (if you are testing a set of classes).

Related Topics

About Static Analysis

Performing Static Analysis

Static Analysis Tutorial

About Dynamic Analysis

Dynamic analysis involves testing a class with actual inputs. To perform dynamic analysis, jtest! automatically generates, executes and evaluates test cases, and, where applicable, executes and evaluates user-defined test cases.

jtest! uses dynamic analysis to perform white-box testing, black-box testing, and regression testing.

Related Topics

Performing Dynamic Analysis

Customizing Dynamic Analysis

Performing Dynamic Analysis

By default, jtest! is configured to perform dynamic analysis. Each time you test a class or set of classes, all appropriate dynamic analysis tests (as well as static analysis) will be performed automatically.

Results

If you tested a project, results are displayed in the **Class Name> Errors Found> Dynamic Analysis** branch of the Project UI's Results panel. To learn more about this panel's branches and available options, see Understanding the Results Panel and Understanding and Customizing Project Test Results.

If you tested a single class, results are displayed in the **Dynamic Analysis** branch of the Errors Found panel. To learn more about this panel's branches and available options, see Understanding the Errors Found Panel and Understanding and Customizing Class Test Results.

Related Topics

About Dynamic Analysis

Customizing Dynamic Analysis

Customizing Dynamic Analysis

Dynamic Analysis can be customized by suppressing dynamic analysis error messages, and by modifying Class, Project, and Global Test Parameters.

The parameters under **Test Case Generation** control the generation of both the automatic test cases (the ones that jtest! generates automatically) and the user-defined test cases.

The parameters under **Test Case Execution** control the execution of all the test cases.

The parameters under **Test Case Evaluation** control the evaluation of all the test cases. Note that the evaluation is performed on both the automatic and user-defined test cases. For example, if **Perform Automatic Regression Testing** is selected, automatic regression testing is performed for both the automatic and the user-defined test cases.

Related Topics

About Dynamic Analysis

Performing Dynamic Analysis

Setting an Object to a Certain State

In some cases, you may want to set up an initial state prior to testing a class. For example, suppose that a class is used as a global object containing static member variables accessible by any other project within the application. When jtest! tests an object that uses this static member variable, a NullPointer Exception will result because the variable has not been set. This problem can be solved by giving jtest! initialization code.

If you want to add code that will be executed before any class is tested (e.g. code that is used to setup and initialize the class), perform the following procedure:

1. In the Class Test Parameters window, open **Dynamic Analysis> Test Case Generation> Common**.
2. Double-click the **Static Class Initialization** node.
3. Enter the initialization code in the Class Initialization Code window.
4. (Optional) If you want to check the method, choose **Save & Check** from the **Options** menu.
5. To save the modification, choose **Save** from the **Options** menu.
6. To exit the Class Initialization Code window, choose **Quit** from the **Options** menu.

About White-Box Testing

White-box testing checks that the class is structurally sound. It doesn't test that the class behaves according to the specification, but instead ensures that the class doesn't crash and that it behaves correctly when passed unexpected input.

White-box involves looking at the class code and trying to find out if there are any possible class usages that will make the class crash (in Java this is equivalent to throwing an uncaught runtime exception).

jtest! completely automates white-box testing. jtest! executes the class under test using a symbolic virtual machine (a virtual machine that uses mathematical equations instead of numbers to internally represent the variable states). While executing each bytecode, it checks if there are solutions to the equations that will make the bytecode throw an uncaught runtime exception. If it finds a solution it reports an error for it. The equations are also solved to find test cases that execute as many branches in the code as possible.

Related Topics

Performing White-Box Testing

Setting an Object to a Certain State

White-Box Testing Tutorial

Performing White-Box Testing

By default, jtest! is configured to perform white-box testing. Each time you test a class or set of classes, white-box testing, all other appropriate dynamic analysis tests, and static analysis will be performed automatically.

Results

If you tested a project, results are displayed in the **Class Name> Errors Found> Uncaught Runtime Exceptions** branch of the Project UI's Results panel. To learn more about this panel's branches and available options, see Understanding the Results Panel and Understanding and Customizing Project Test Results.

If you tested a single class, results are displayed in the **Uncaught Runtime Exceptions** branch of the Errors Found panel. To learn more about this panel's branches and available options, see Understanding the Errors Found Panel and Understanding and Customizing Class Test Results

Related Topics

About White-Box Testing

Setting an Object to a Certain State

White-Box Testing Tutorial

About Black-Box Testing

Black-box testing checks that the class behaves according to the specification, meaning that the class produces the correct output (outcomes) for each input.

In general, any black-box testing tool requires the user to specify a set of inputs to try, along with the correct outputs for those inputs. The tool then runs the inputs automatically and checks that the correct outputs are generated.

Black-box testing becomes much simpler when you use jtest!

jtest! automatically generates a set of sophisticated inputs; jtest! analyzes the class bytecodes and provides a minimal set of inputs that produce as much coverage as possible for the class.

In addition to using these automatically generated inputs, you can also specify an additional set of inputs in the standard fashion: by listing, for each class method and argument, the inputs that jtest! should try.

jtest! then executes all the inputs and provides you with the outcomes for those inputs, organized in a simple tree representation. The user can then see the outcomes and verify them with one click of a button, so when jtest! performs subsequent tests on this class, it will be able to notify you when specification and regression testing errors occur.

Note: To perform black-box testing with jtest, you must have the JDK compiler on your PATH. The 'javac' command is the compiler and it is usually located under the bin directory of most JDK installations.

Related Topics

Performing Black-Box Testing (From Class Testing UI)

Performing Black-Box Testing (From Project Testing UI)

Adding Inputs

Specifying Imports

Setting an Object to a Certain State

Black-Box Testing Tutorial

Performing Black-Box Testing (From Class Testing UI)

By default, jtest! is configured to perform black-box testing as long as outcomes from automatically generated and/or user-defined test cases have been validated.

Each time you test a class or set of classes after validating test case outcomes, black-box testing, all other appropriate dynamic analysis tests, and static analysis will be performed automatically.

To add user-defined inputs

1. Test the class automatically with jtest! (i.e., simply select the class you want to test and click the **Start** icon).
2. Review the test cases that jtest! generated automatically in the View Test Cases window (accessible by clicking the **View** button in the Class Testing UI toolbar), then decide what additional test cases you want to add.
3. Add any additional inputs.
   To add an input, perform the following steps:
   - Open the Class Test Parameters window.
   - In the Class Test Parameters window, go to **Dynamic Analysis> Test Case Generation> User Defined> Method Inputs** to view a list of all methods in the class.
   - Open the node associated with the method whose inputs you want to define.
   - Right-click the argument you want to define an input for, then choose **Add Input Value** from the shortcut menu.
   - In the text field of the box that opens, type the input that you want to use, then press **Enter** to save this value.
     Note: You can also add constants or methods in the manner described in Adding Inputs. In addition, you can specify imports shared by all the code used in the specification, or add class initialization code to set an object to a certain state.
4. Retest the class by clicking the **Start** button. jtest! will run both the automatic test cases and the user-defined test cases.
5. Review the outcomes for the test cases in the View Test Cases window (to access this window, click the **View** button in the Class Testing UI toolbar.)

To evaluate outcomes

1. Indicate whether or not the outcome for each test case is correct by right-clicking the appropriate outcome, and choosing **Correct** (if the listed out-

come is the expected outcome), **Incorrect** (if the listed outcome is not the expected outcome), **Unknown** (if you don't know how the listed outcome compares to the expected outcome), or **Ignore** (if you want jtest! to ignore the listed outcome) from the shortcut menu. To indicate that all outcomes for a test case are correct, right-click the appropriate **Outcomes** leaf and choose **Set All Correct** from the shortcut menu.

Now every time jtest! is run on that class, jtest! will automatically run that user-defined test case and will check that the outcomes are correct.

Results

The specification errors that jtest! finds are reported under the **Specification and Regression Errors** header in the Errors Found panel of the Class Testing UI.

The Errors Found panel's contents and functionality are described in Understanding the Errors Found Panel.

The Errors Found panel also provides you with a variety of ways to gain additional information about the tests performed and test results and to customize what results are reported the next time this test is run. For more information about these options, see Understanding and Customizing Project Test Results.

Related Topics

About Black-Box Testing

Performing Black-Box Testing (From Project Testing UI)

Adding Inputs

Specifying Imports

Setting an Object to a Certain State

Black-Box Testing Tutorial

Performing Black-Box Testing (From Project Testing UI)

By default, jtest! is configured to perform black-box testing as long as outcomes from automatically generated and/or user-defined test cases have been validated.

Each time you test a class or set of classes after validating test case outcomes, black-box testing, all other appropriate dynamic analysis tests, and static analysis will be performed automatically.

To add and validate test cases:

To add user-defined inputs

1. Test the set of classes automatically with jtest! (i.e., simply select the set of class you want to test and click the **Start** icon).
2. For each class you want to add inputs for, review the test cases that jtest! generated automatically. Test cases can be viewed by right-clicking the class name in the Results for All Classes panel, then choosing **View Test Cases** from the shortcut menu. Review the test cases in the View Test Cases window, then decide what additional test cases you want to add.
3. Add any additional test cases.
    - Open the Class Test Parameters window.
    - In the Class Test Parameters window, go to **Dynamic Analysis> Test Case Generation> User Defined> Method Inputs** to view a list of all methods in the class.
    - Open the node associated with the method whose inputs you want to define.
    - Right-click the argument you want to define an input for, then choose **Add Input Value** from the shortcut menu.
    - In the text field of the box that opens, type the input that you want to use, then press **Enter** to save this value.
      Note: You can also add simple and complex inputs in the manner described in Adding Inputs. In addition, you can specify imports shared by all the code used in the specification, or add class initialization code.
4. Retest the class by clicking the **Start** button. jtest! will run both the automatic test cases and the user-defined test cases.
5. Review the outcomes for the test cases in the View Test Cases window To access this window, right-click the class name in the Results for All Classes window, then choose **View Test Cases** from the shortcut menu.

To evaluate outcomes

1. Indicate whether or not the outcome for each test case is correct by right-clicking the appropriate outcome, and choosing **Correct** (if the listed outcome is the expected outcome), **Incorrect** (if the listed outcome is not the expected outcome), **Unknown** (if you don't know how the listed outcome compares to the expected outcome), or **Ignore** (if you want jtest! to ignore the listed outcome) from the shortcut menu. To indicate that all outcomes for a test case are correct, right-click the appropriate **Outcomes** leaf and choose **Set All Correct** from the shortcut menu.

Now every time jtest! is run on that class, jtest! will automatically run that user-defined test case and will check that the outcomes are correct.

Results

The specification errors that jtest! finds are reported under the **Specification and Regression Errors** header in the Results panel of the Project Testing UI.

The Results panel's contents and functionality are described in Understanding the Results Panel.

The Results panel also provides you with a variety of ways to gain additional information about the tests performed and test results and to customize what results will be reported the next time this test is run. For more information about these options, see Understanding and Customizing Project Test Results.

Related Topics

About Black-Box Testing

Performing Black-Box Testing (From Class Testing UI)

Adding Inputs

Specifying Imports

Setting an Object to a Certain State

Black-Box Testing Tutorial

Adding Inputs

jtest! makes it easy to add your own test case inputs. You can store constants and methods to be used as inputs for black-box testing in two input repositories:

1. **Local:** Contains inputs specific to the class under test. Configured in Class Test Parameters window.
2. **Global:** Contains inputs available to any class under test. Configured in Global Test Parameters window.

You can also add primitive inputs directly to a method's node in the Class Test Parameters window.

Note: All inputs are added to arguments in the Class Test Parameters window. In the Class Testing UI, you can open this window by clicking the **Params** button. To open this window from the Project Testing UI, right-click the **[Class Name]** node associated with the class you want to add inputs to, and choose **Edit Class Test Parameters** from the shortcut menu.

Adding Primitive Inputs

You can use either of the following procedures to define your own inputs to methods under test.

Adding Primitive Inputs to Your Test

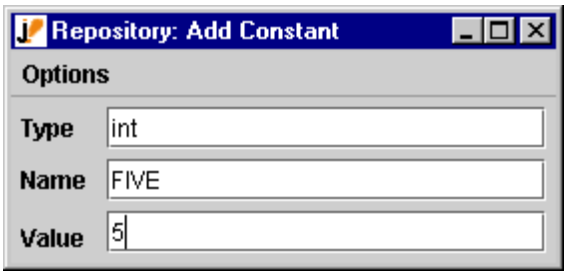
To add primitive inputs to your test:

1. Open the Class Test Parameters window.
2. In the Class Test Parameters window, go to **Dynamic Analysis> Test Case Generation> User Defined> Method Inputs** to view a list of all methods in the class.
3. Open the node associated with the method whose inputs you want to define.
4. Right-click the argument you want to define an input for, then choose **Add Input Value** from the shortcut menu.
5. In the text field of the box that opens, type the input that you want to use, then press **Enter** to save this value.

Adding Constants to an Inputs Repository

To add a constant to an Inputs Repository:

1. Right-click **Inputs Repository** in the Global or Class Test Parameters window. A shortcut menu will open.
2. Choose **Add Constant** from the shortcut menu. The Add Constant window will open.
3. In the Add Constant window, enter the type of the constant (e.g. int) in the **Constant** field, the name of the constant (e.g. FIVE) in the **Name** field, and the value of the constant (any valid Java expression-- e.g., 5) in the **Value** field.



4. Choose **Options> Save**.
5. Choose **Options> Quit**.

Adding Non-Primitive Inputs

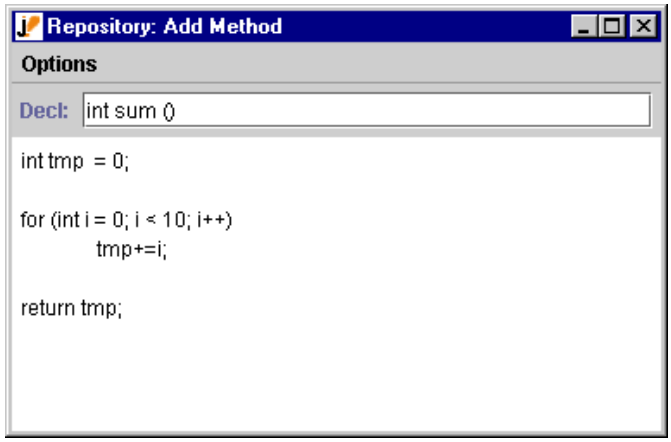
You can also use non-primitive (object type) inputs as method parameters. To do this:

- Define a method which will instantiate and set up the object input as you'd like, and add it to an Inputs Repository. (see Adding Methods to an Inputs Repository for instructions)
- Add the input argument for the argument you want to set by adding the method you defined in the Inputs Repository to that argument. (see Adding Inputs From a Repository for instructions)

Adding Methods to an Inputs Repository

To add a method to an Inputs Repository:

1. Right-click **Inputs Repository** in either the Global or Class Parameters tree. A shortcut menu will open.
2. Choose **Add Method** from the shortcut menu. The Add Method window will open.
3. In that window, define a method that creates and returns the desired input values.



4. Enter the method declaration (e.g. int sum()) in the **Decl** field.
5. (Optional) If you want to check the method, choose **Save & Check** from the **Options** menu.
6. Choose **Options> Save**.
7. Choose **Options> Quit**.

Adding Inputs From a Repository

When you want to add inputs from either repository to an argument:

1. In the Class Test Parameters window, right-click the node associated with the argument you want to add an input value to. A shortcut menu will open.
2. From the shortcut menu, choose either **Add From Local Repository** (if the input is in the local repository), or **Add From Global Repository** (if the input is in the global repository), then choose the desired input.

Related Topics

About Black-Box Testing

Performing Black-Box Testing (From Class Testing UI)

Performing Black-Box Testing (From Project Testing UI)

Specifying Imports

Setting an Object to a Certain State

Black-Box Testing Tutorial

Specifying Imports

To specify import statements shared by all the code used in the test specification:

1. In the Class Test Parameters window, open **Dynamic Analysis> Test Case Generation> Common**.
2. Double-click the **Imports** node. The Imports window will open.
3. Enter the import statement in the Imports window.
   Example:
   ```
   import java.util.Vector;
   import java.awt*;
   ```
4. (Optional) If you want to check the method, choose **Save & Check** from the **Options** menu.
5. From the **Options** menu, choose **Save**.
6. From the **Options** menu, choose **Quit**.

Related Topics

About Black-Box Testing

Performing Black-Box Testing (From Class Testing UI)

Performing Black-Box Testing (From Project Testing UI)

Adding Inputs

Setting an Object to a Certain State

Black-Box Testing Tutorial

About Regression Testing

Regression testing checks that the class behavior doesn't change. Regression testing gives you the peace of mind of knowing that the class doesn't break when the code is modified.

Regression testing is very similar to black-box testing in that both check whether certain inputs generate certain outputs. Whereas black-box testing is performed to verify that the class performs according to specification, regression testing aims to assure that the class *continues* to perform according to specification after it has been modified.

jtest! provides automatic regression testing. Even if you don't specify what the correct outcomes are, jtest! remembers the outcomes from previous runs and compares them every time the class is tested and reports an error for any outcome that changes.

Related Topics

Performing Regression Testing

Regression Testing Tutorial

Performing Regression Testing

By default, jtest! is configured to perform regression testing. Each time you test a class or set of classes, regression testing will be performed automatically.

Two steps are required to perform regression testing:

1. Restore previously saved test parameters by choosing **File> Open**, then typing or selecting the test parameters you want to restore.
   Note: You can also open recent parameters by choosing **File> Open Recent> [File Name]**.
2. Run the test by clicking the **Start** button.

Results

If you tested a project, results are displayed in the **Class Name> Errors Found> Specification and Regression Errors** branch of the Project UI's Results panel. To learn more about this panel's branches and available options, see Understanding the Results Panel and Understanding and Customizing Project Test Results.

If you tested a single class, results are displayed in the **Specification and Regression Errors** branch of the Errors Found panel. To learn more about this panel's branches and available options, see Understanding the Errors Found Panel and Understanding and Customizing Class Test Results

Related Topics

About Regression Testing

Regression Testing Tutorial

Saving and Restoring Tests Parameters

Saving Parameters

If you save test parameters, you will be able to instantly restore the precise testing circumstances that you used for a previous test. This lets you accurately repeat a test to see if class modifications caused class behavior to change and/or introduced errors (Regression Testing).

Whenever jtest! tests a class it automatically saves the class test parameters in the file shown in the status bar. Whenever jtest! tests a set of classes, it automatically saves project test parameters for the project as a whole, as well as class test parameters for each class contained in the project.

You can also save parameters under a different name by choosing **Save As** from the **File** menu in the appropriate UI (use the Project Testing UI to save project test parameters, and the Class Testing UI to save class test parameters).

Restoring Test Parameters

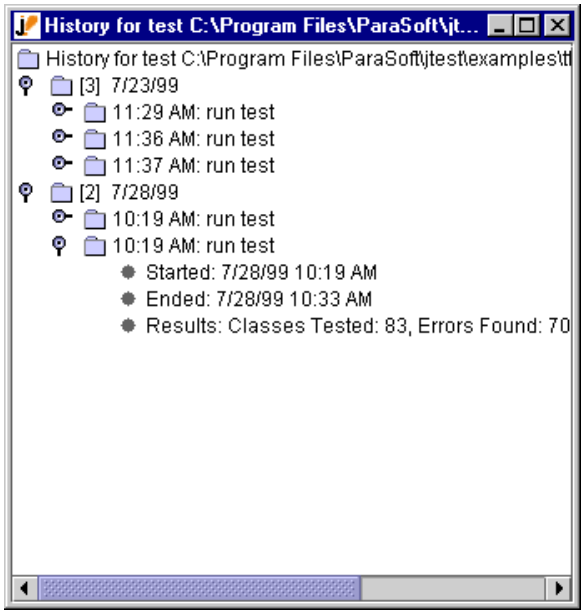
To restore parameters from a previous test (for example, to perform regression testing), simply choose **File> Open** from the appropriate UI (use the Class Testing UI to restore class test parameters; use the Project Testing UI to restore project test parameters). In the dialog box that opens, type or select the parameter file that you want to restore, then jtest! will open the specified test parameters.

You can also open recent parameters by choosing **File> Open Recent> [File Name]**.

Viewing Test History

To view a record of all previous Project Test runs (including test start and end time, as well as a brief summary of test results) for the current project test, click the **History** button in the Project Testing UI toolbar.

To view a record of all previous Project Test runs (including test start and end time, as well as a brief summary of test results), right-click the **History** button in the Project Testing UI toolbar, and choose **Global History** from the shortcut menu.


History for test C:\Program Files\ParaSoft\jt...
History for test C:\Program Files\ParaSoft\jtest\examples\tt
[3] 7/23/99
11:29 AM: run test
11:36 AM: run test
11:37 AM: run test
[2] 7/28/99
10:19 AM: run test
10:19 AM: run test
Started: 7/28/99 10:19 AM
Ended: 7/28/99 10:33 AM
Results: Classes Tested: 83, Errors Found: 70

Both history windows have identical functionality. To remove a record from the test history, right-click the appropriate record, and choose **Delete** from the short-cut menu.

To view the log, report, or results for a specific test, right-click the record of the run that you want to view information for, and choose the appropriate command from the shortcut menu.

You can also delete all entries by right-clicking the **History** node and choosing **Delete All Entries** from the shortcut menu.

Viewing Context-Sensitive Help

jtest! has context-sensitive help topics associated with almost every option, command, and UI component.

To view context-sensitive help topics:

1. Enable context-sensitive help by clicking the **Context Help** button in either UI's toolbar.
2. Move your cursor over the item you want to learn more about. If a context-sensitive help topic is available for this element, that topic will then open.

Viewing, Editing, or Compiling A Source

Viewing the Source of A Violation

You can view the source of a violation, with the problematic line highlighted, by double-clicking the file/line information for the error in the Errors Found panel (in Class Testing UI) or in the Errors Found branch of the lower Results panel (in Project Testing UI). Another way to view the source is to right-click that same error message and choose **View Source** from the shortcut menu.

Editing the Source of A Violation

You can also edit the source of a violation from within jtest! To edit the source of a violation, right-click the file/line information for the violation message in the Errors Found panel (in Class Testing UI) or in the Errors Found branch of the lower Results panel (in Project Testing UI), and choose **Edit Source** from the shortcut menu.

Viewing, Editing, and Compiling Any Source

In addition, you can view, edit, and compile the source of *any* class under test (whether or not it contains a violation), via the Class Testing UI. Before you perform any of these actions, you must first open the class whose source you want to view, edit, or compile in the Class Testing UI. When your class is open in the Class Testing UI, you can perform any of the following actions:

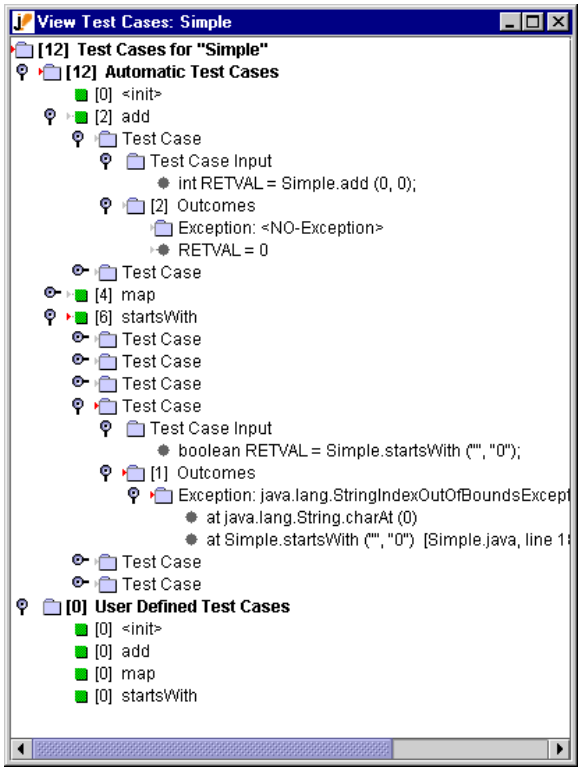
- **View class source in jtest!'s Source Viewer:** Right-click the **Source** button and choose **View Class Source**.
- **Edit class source in Notepad or other editor:** Right-click the **Source** button and choose **Edit Class Source**. Notepad is used as the default editor. To use a different editor, choose **Editor** from from **Customize** menu, and enter your preferred editor in the dialog box that opens.
- **Compile the current class:** Right-click the **Source** button and choose **View Class Source**.

Viewing and Validating Test Cases

The View Test Cases window displays and lets you validate the test cases that jtest! used for Dynamic Analysis.

To open this window from the Class Testing UI, click the **View Test Cases** button.

To open this window from the Project Testing UI, right-click either the **[Class Name]** node or the **[Class Name]> Test Progress> Dynamic Analysis> Number of Test Cases Executed** node, and choose **View Test Cases** from the shortcut menu.



About the View Test Cases window

The View Test Cases window contains the following leafs:

Test cases for [classname]

Contains test cases generated and executed by jtest! in the last test for this class.

Automatic Test Cases

Contains test cases generated automatically by jtest! Of all the test cases generated, only the test cases that do something new (e.g., increase coverage, throw a new exception, etc.) are shown.

[method name]

Contains test cases for this method.

Test Cases

Contains all the information for a test case.

Test Case Input

Input that defines the test case.

The input for automatic test cases is the calling sequence.

The input for user defined test cases is the input for each argument.

Outcomes

Outcomes for this test case. Verify if the outcomes are correct or incorrect according to the class specification and set their state using the shortcut menus.

When the outcome is an object, jtest! automatically chooses the toString method to show its state.

If a method called jtestInspector is defined for the object's class, jtest! will only use the return value of this method to show the object state.

If no toString or jtestInspector methods are defined, jtest! will heuristically choose some public instance methods for that object to show its state.

If the method under test is a static method, jtest! will heuristically choose public static methods to show the class state. If the methods jtest! chose are not enough, declare a static method called sjtestInspector for the class. jtest! will use the return value of this method to show the object class.

[n]= number of outcomes for this test case.

Exception

Indicates whether an exception occurred, and, if so, what type of exception occurred.

If an exception was suppressed, you can see the reason for the suppression by right-clicking the exception message node and choosing **Why Suppressed?** from the shortcut menu.

User Defined Test Cases

Contains test cases generated from user-defined input.

[method name]

Contains test cases for this method.

Test Case

Contains all the information for a test case.

Test Case Input

Input that defines the test case.

The input for automatic test cases is the calling sequence.

The input for user defined test cases is the input for each argument.

Outcomes

Outcomes for this test case. Verify if the outcomes are correct or incorrect according to the class specification and set their state using the shortcut menus.

When the outcome is an object, jtest! automatically chooses the toString method to show its state.

If a method called jtestInspector is defined for the object's class, jtest! will only use the return value of this method to show the object state.

If no toString or jtestInspector methods are defined, jtest! will heuristically choose some public instance methods for that object to show its state.

If the method under test is a static method, jtest! will heuristically choose

public static methods to show the class state. If the methods jtest! chose are not enough, declare a static method called sjtestInspector for the class. jtest! will use the return value of this method to show the object class.

[n]= number of outcomes for this test case.

Exception

Indicates whether an exception occurred, and, if so, what type of exception occurred. When an exception occurs, stack trace information can be displayed by opening this node.

The color of the arrow to the left of a leaf has the following meaning:

- **green:** the outcome is correct, or has been validated as correct by the user.
- **red:** the outcome is incorrect or has been validated as incorrect by the user.
- **gray:** the outcome status is unknown.
- **no arrow:** the user has specified to ignore this outcome.

If the **Perform Automatic Regression Testing** flag is selected, jtest! will assume that gray outcomes are correct and will report an error if the outcome changes.

In this window, the outcome is marked as incorrect if it is different from the one in the Specification and Regression Test Cases branch of the Errors Found panel (in Class Testing UI) or Results panel (in Project Testing UI). When more than one test case outcome differs, only one of them is marked as an error and only that one is reported as an error in the Errors Found panel or Results panel.

Validating Outcomes

Indicate whether or not the outcome for each test case is correct by right-clicking the appropriate outcome node, and choosing **Correct** (if the listed outcome is the expected outcome), **Incorrect** (if the listed outcome is not the expected outcome), **Unknown** (if you don't know how the listed outcome compares to the expected outcome), or **Ignore** (if you want jtest! to ignore the listed outcome) from the shortcut menu.

To indicate that all outcomes for a test case are correct, right-click the appropriate **Outcomes** leaf and choose **Set All Correct** from the shortcut menu.

To ignore an entire test, right-click the appropriate **Test Case** node, and choose **Ignore this Test Case** from the shortcut menu. To tell jtest! not to ignore a test case you previously told jtest! to ignore, right-click the appropriate **Test Case** node, and choose **Do Not Ignore this Test Case** from the shortcut menu.

To remove an entire test case, right-click the appropriate **Test Case** node, and choose **Delete** from the shortcut menu.

Viewing A Report of Results

jtest! generates the following types of reports:

- Single Class Report
- Project Report
- Detailed Project Report
- Summary Project Report

These reports use the standard JNI 1.1 specification to identify methods.

Single Class Report

After performing a test on a single class, jtest! will generate a Single Class Report of the testing session. The report is generated for the single class that was just tested and whose name appears in the Class Name panel.

The Single Class Report is accessed by clicking the **Report** button in the Class Testing UI toolbar.

This report contains, among other information, the annotated source code for the tested class. This may be used to determine what lines jtest! tested and what lines it did not test. Lines of code that were not tested are marked with a “>” at the beginning of the line; absence of the “>” indicates that a line was tested.

By default, jtest! does not show the source of every accessed class unless told to do so. Use the **Customize> Report File> Show All Classes Accessed** option to tell jtest! to display the annotated source of each class accessed by the class under test.

Project Report

The Project Report is accessed by clicking the **Report** button in the Project UI toolbar, or by using -report at the command line.

This report is more detailed than the Summary Project Report, but less detailed than the Detailed Project Report. This report contains project test parameters as well as all essential details about each error available in the Results panel.

All project reports only contain information on the classes and errors available in the Results panel when the **View Results** button was clicked. To limit the

classes and errors contained in your report, display only the desired classes and errors in the Results panel before you click the **View Results** button.

Detailed Project Report

The Detailed Project Report is accessed by right-clicking the **Report** button in the Project UI toolbar and choosing **View Detail Report** from the shortcut menu; it can also be obtained by using -detail_report in the command line.

This is the most detailed of the three available project reports. This report contains project test parameters, class test parameters, and all results information available in the Results panel.

All project reports only contain information on the classes and errors available in the Results panel when the **View Results** button was clicked. To limit the classes and errors contained in your report, display only the desired classes and errors in the Results panel before you click the **View Results** button.

Summary Project Report

After a set of classes has been tested, jtest! will generate an Summary Project Report.

The Summary Project Report is accessed by right-clicking the **Report** button in the Project UI toolbar and choosing **View Summary Report** from the shortcut menu; it can also be obtained by using -summary_report in the command line.

This is the least detailed of the three available project reports. This report contains the test name and a one line report of each error available in the Results panel; if there are no errors, this report contains only the test name.

All project reports only contain information on the classes and errors available in the Results panel when the **View Results** button was clicked. To limit the classes and errors contained in your report, display only the desired classes and errors in the Results panel before you click the **View Results** button.

This report can be used to compare the output of jtest! when calling it from a test harness.

Viewing the Contents of A Results Directory

jtest! automatically saves results for each test in a Results directory.

Viewing Results of Project Tests

To view all project test results stored in a results directory, click the **Results** button in the Project UI toolbar.

To view only the results from the last project test run, right-click the **Results** button in the Project UI toolbar and choose **View Results From Last Test Run** from the shortcut menu.

Viewing Results of Class Tests

To view the results of all class tests stored in a results directory, choose **Open** from the Results menu in the Class Testing UI, then type or select the directory where the desired results are saved. Or, if you want to view results from a recent test, choose **Open Recent** from the Results menu in the Class Testing UI, then choose the desired results filename from the list of results files. The file will open in the Results UI.

Understanding Test Results

The Results UI's Results panel and the Results panel of the Project Testing UI share the same structure and functionality. For more information about this Results panel's contents and basic functionality, see Understanding the Results Panel; to learn about commands in this panel that increase your understanding of the results and customize results, see Understanding and Customizing Class Test Results.

Changing the Results Directory

To change the directory where results files are stored:

1. Open the Parameters window in which you want to change the Results Directory parameter (this parameter appears in Global, Project, and Test Parameter windows).

2. In the Parameter window open **Common Parameters> Directories> Results**.
3. Right-click **Results**. A shortcut menu opens.
4. Choose **Edit** from the shortcut menu.
5. In the text box that opens, enter the desired directory.

Running jtest! in Batch Mode

You can run jtest! in batch mode by performing the following steps:

1. Set up a test through the Project UI.
   - Start jtest!.
   - Click the **Project** button in the Class Testing UI.
   - In the Project UI's Controls panel, indicate where you want jtest! to start finding and testing classes, then click the **Start** button.
   - Choose **File> Save** to save the project test as a test parameters file (for example, enter `MyTest.ptp`)

2. Set up command line jtest!
   - cd to the jtest! installation directory.
   - Run the setenv.bat program by typing
     `setenv.bat`
     at the prompt.

3. Run the test from the command line.
   - At the prompt, enter your command. For example, you could enter the following command at the prompt:
     `jtestgui -summary_report summary.rpt -test MyTest.ptp`
     This would run the MyTest.ptp project setup file, then generate a summary report in the current working directory (summary.rpt).
     If you wanted to perform this same action in “silent mode” (without displaying the jtest! UI), you would enter the following command at the prompt:
     `jtestgui -nogui -summary_report summary.rpt -test MyTest.ptp`

To view available command line options, type

```
jtestgui -help
```

after you have run setenv.bat.

Customizing Test Parameters

The testing done by jtest! is highly customizable. Test parameters can be customized at three levels:

- Global Test Parameters (apply to all tests)
- Project Test Parameters (apply to a specific project, or set of classes)
- Class Test Parameters (apply to a specific class)

For details on each of these types of parameters, follow the above links.

Parameters that appear at more than one level

When setting parameters, be aware that several parameters appear at more than one level. For example the parameter **Static Analysis> Rules> Severity Levels** appears in all parameters (Global, Project and Class).

When testing a class, jtest! uses the value in the Class Test Parameters. If a value is set to "default," the value of the current parent parameter is used. In this case, the actual value that will be used in shown in the icon or in parentheses.

When creating a new test, the value in the parent parameter is used. For example, when creating an Project test, the value of the flag in the Global Test Parameters is used as the initial value.

Customizing Reporting of Violations

To customize jtest! so that it only reports the errors relevant to your project, you can suppress the reporting of any static analysis warning messages and uncaught runtime exceptions that you do not want displayed.

Related Topics

Static Analysis Suppressions

Dynamic Analysis Suppressions

Static Analysis Suppressions

You can suppress the reporting of rule violation messages in two ways:

- Adding a suppression from the Errors Found panel or Results panel (recommended method).
- Adding a suppression directly to the Static Analysis Suppressions Table.

However, because the rule violation messages will change from class to class, the best way to "suppress" the reporting of warning messages from particular rules or sets of rules is by turning rules (or rule categories) on and off in the manner described in Customizing Static Analysis.

To suppress a particular warning message that appears in the Errors Found panel or Results panel, right-click the message that you want suppress, and choose **Suppress** from the shortcut menu. This action will add that specific warning message to the Static Analysis Suppressions Table.

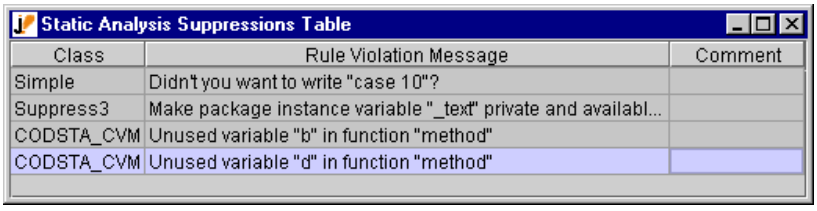
Opening the Suppressions Table

To reach the Suppressions Table, double-click the **Suppressions Table** node in the **Static Analysis** branch of the Global Test Parameters tree.

Adding a Suppression to the Static Analysis Suppressions Table

1. Right-click any area of the table. A shortcut menu will open.
2. From the shortcut menu, choose **Add New Suppression**. An empty table entry will open.
3. In the empty table entry, enter the desired options. The format of a suppression should be as follows:

<fully qualified classname><exact message from static analysis>



Removing a Suppression

1. Right-click the table entry. A shortcut menu will open.
2. Choose the **Delete** menu item from the shortcut menu.

Note: To delete a list of suppressions, right-click drag from the first suppression to the final one, then release the mouse button. A shortcut menu will open. Choose **Delete** from this menu.

Viewing the Results

Suppressions in this list are applied when the results are displayed. Results are displayed after you click the **Start** button, generate the report file, or view the Results UI. After you have changed the suppressions, you can see an updated report of violations by viewing the results again.

Dynamic Analysis Suppressions

The best way to stop jtest! from displaying certain uncaught runtime exceptions is to right-click that message in the Errors Found panel of the Class Testing UI or in the Results panel of the Project Testing UI, then choose one of the following commands from the shortcut menu:

- **Not an Error:** Choose this command to indicate that a reported uncaught runtime exception is not an error, but rather the correct behavior of the class (for that input).
- **Ignore:** Choose this command to tell jtest! to ignore the uncaught runtime exceptions for this input. If this is selected, the outcome will not be used for comparisons when searching for specification or regression errors. Also, no uncaught runtime exceptions will be reported for this input.

You can also suppress the reporting of uncaught runtime exceptions in two ways:

- Adding a suppression from the Errors Found panel or Results panel.
- Adding a suppression directly to the Dynamic Analysis Suppressions Table.

To suppress a particular exception that appears in the Errors Found panel or Results panel, right-click the exception that you want suppress, and choose **Suppress** from the shortcut menu. This action will add that specific exception to the Dynamic Analysis Suppressions Table.

Additionally, you can enable or disable checking for certain types of exceptions in the **Pre-filtering Suppressions Categories** node (located under **Dynamic Analysis> Test Case Execution**) of the Global, Project, or Class Test Parameters.

The Dynamic Analysis Suppressions Table lets you create new suppression categories for uncaught runtime exceptions. You can use this option to suppress reporting of exceptions by type, as well as by the class and name where they occur.

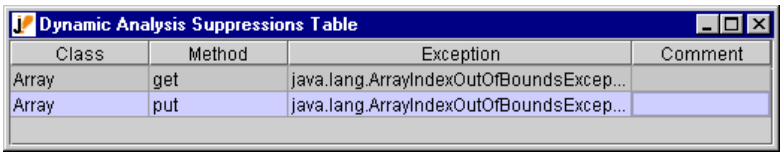
Opening the Suppressions Table

To reach the Suppressions Table, double-click the **Suppressions Table** node in the **Dynamic Analysis** branch of the Global Test Parameters tree.

Adding a Suppression to the Dynamic Analysis Suppressions Table

1. Right-click any area of the table. A shortcut menu will open.
2. Choose **Add New Suppression** from the shortcut menu. An empty table entry will open.
3. In the empty table entry, enter the exception, the method that throws it, or the class that declares the method that you want to suppress.

Important: Make sure that you type all values exactly as they appear in the jtest! UI.



Removing a Suppression

1. Right-click the table entry. A shortcut menu will open.
2. Choose **Delete** from the shortcut menu.

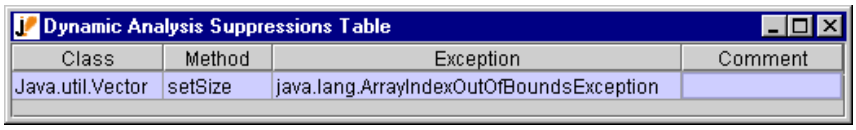
Note: To delete a list of suppressions, right-click drag from the first suppression to the final one, then release the mouse button. A shortcut menu will open. Choose **Delete** from the shortcut menu.

Viewing the Results

Suppressions in this list are applied when the results are displayed. Results are displayed after you click the **Start** button, generate the report file, or view the Results UI. After you have changed the suppressions, you can see an updated report of exceptions by viewing the results again.

Suppressing Specific Exceptions by Class, Method, and Exception Type

To suppress specific exceptions by class, method, and exception type, enter the information in the appropriate fields. For example, to suppress the ArrayIndex-OutOfBoundsException from the 'setSize' method of java.util.Vector, use:

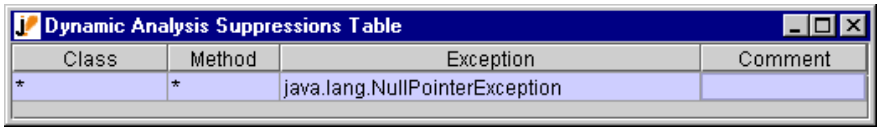


Suppressing Exceptions by Class

To suppress exceptions by class, put the classname in the **Class** field by double clicking that field and then entering the fully qualified classname. For example, to suppress all exceptions from the class java.util.Vector, use:

```
java.util.Vector | * | *
```

Use the '*' (asterisk) symbol to match any letter. For example, to suppress all NullPointerExceptions use:



Suppressing Exceptions by Method

To suppress exceptions by method, put the method name in the **Method** field by double clicking that field and them entering the method name. To suppress exceptions from a single method belonging to a set of overloaded methods, specify the method by including its signature enclosed in parentheses. Method signatures should follow the JNI specification from the JDK (minus the return type). For example, to suppress exceptions from the following method

```
(int n, String s, int []ia)
```

use the following as the method signature:

```
(ILJava/lang/String;[I)
```

Customizing System Settings

Use commands found in the **Preferences** menu of the Class Testing UI or Project Testing UI menu bar to customize the following aspects of jtest!:

- **Editor:** Opens a dialog box that allows you to determine what editor is invoked when you view report files and edit your source. If the editor command includes white-space, be sure to enclose the command in quotation marks. To represent the file parameter and the line number parameter, use the special tokens $FILE and $LINE. (**Preferences> Editor**)
- **Starting UI:** Determines what UI opens by default when jtest! is started. (**Preferences> Starting UI**)
- **Title Bar Background Color:** Determines the title bar's background color.(**Preferences> Title Bar Background Color**)
- **Context Help Font:** Determines what font type and size is used to display context-sensitive help. (**Preferences> Help Font Size**)
- **Look and Feel:** Changes the look and feel of the jtest!'s UI. (**Preferences> Look and Feel**)
- **Tips:** Reactivates or deactivates context-sensitive tips. (**Preferences> Tips**)
- **Report File:** Customizes report file characteristics. (**Preferences> Report File**)
- **Show All Classes Accessed:** Prompts jtest! to annotate all sources (when displaying the report file) for each user class accessed during its testing.(**Preferences> Report File> Show All Classes Accessed**)

jtest! UI Overview

jtest! has two available UIs:

- Class Testing UI: Area to test a single class or view results of a single class tested as part of a project test.
- Project Testing UI: Area to test a set of classes (from a directory, zip file, or jar file).

By default, the Class Testing UI opens when jtest! is started. To configure jtest! so the Project Testing UI appears when jtest! is started, choose **Starting UI** from the **Customize** menu in either UI, then choose **Project Testing UI**.

Trees

The following features are common to all of jtest!'s trees:

1. Most tree nodes have shortcut menus associated to them; whenever to want to do something with a tree node, check if there is an associated shortcut menu. If context help is active and the cursor is placed on the shortcut menu, jtest! will display the help text for that shortcut menu.
2. A right-click icon will appear when your cursor is placed over a tree node that has a shortcut menu with additional commands. Right-click the node to access the shortcut menu containing the additional commands.
3. All the trees have an extra shortcut menu which is accessed by clicking the right mouse button with the **Control** key pressed.
   This extra shortcut menu contains the following commands:
   - **Find:** Finds strings in the tree.
   - **Print:** Prints the tree.
   - **Show Children:** Completely expands the tree to reveal all children
   - **Check:** Checks the node contents (if the node allows that operation) and/or view any error messages associated to the node.

Cursors

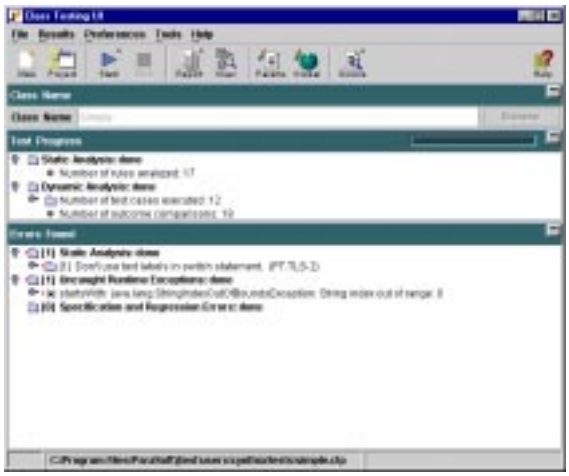
jtest! uses two special cursors to alert you to “hidden” options and/or information:

- **the** **help cursor:** After context-sensitive help has been enabled, this cursor indicates that there is a context-sensitive help topic available for the item the cursor is positioned over.
- **the** **right-click cursor:** This cursor indicates that a shortcut menu is associated with the item the cursor is positioned over. The shortcut menu can be accessed by right-clicking the item.

Class Testing UI

The Class Testing UI allows you to perform and configure tests of single classes, as well as focus on the results of a class tested as part of a project test. Components of this UI are:

- Menu Bar
- Toolbar
- Class Name Panel
- Test Progress Panel
- Errors Found Panel



For information on testing a class in the Class Testing UI, see Testing A Single Class.

Menu Bar

File

Commands in this menu control test session functionality.

- **New Session:** Starts a new session by clearing out any existing setup values or settings.
- **Open:** Allows you to opens an existing test specification; opens a dialog box in which you can specify a test parameters file.
- **Open Recent:** Allows you to open parameters of a recent test. Contains a list of the most recently opened classes; choose a file name from the list to open the associated parameters file.
    - **Clear List:** Clears all items from this list.
- **Save:** Saves the current class test parameters into the test parameters file shown in the status bar.
- **Save As:** Saves the current class test parameters into a user-specified file.
- **Project Testing UI:** Opens the Project Testing UI (used to test multiple classes).
  Note: When the Project Testing UI is open, it takes control of the Class Testing UI.
- **Close UI:** Closes the Class Testing UI. If the Project Testing UI is not open, choosing this command will also close jtest!
- **Exit:** Closes jtest!

Results

Commands in this menu allow you to access results of previous tests.

- **Open:** Opens a Results directory.
- **Open Recent:** Allows you to open results of a recent test. Contains a list of the most recently opened classes; choose a file name from the list to open the associated results file.
    - **Clear List:** Clears all items from this list.

Preferences

Commands in the menu let you customize jtest! system settings.

- **Editor:** Opens a dialog box that allows you to determine what editor is invoked when you view report files and edit your source. If the editor command includes white-space, be sure to enclose the command in quotation

marks. To represent the file parameter and the line number parameter, use the special tokens $FILE and $LINE.

- **Title Bar Background Color:** Determines the title bar's background color.
- **Starting UI:** Determines what UI opens by default when jtest! is started.
- **Context Help Font:** Determines the size and type of the font used to display context-sensitive help text.
- **Look and Feel:** Changes the look and feel of the jtest!'s UIs.
- **Tips:** Contains options that configure context-sensitive tips.
    - **Reactivate All:** Reactivates all context-sensitive tips.
    - **Deactivate All:** Turns off all content=sensitive tips.
- **Report File:** Customizes report file characteristics
    - **Show All Classes Accessed:** Prompts jtest! to annotate all sources (when displaying the report file) for each user class accessed during its testing.

Tools

Commands in this menu access jtest!'s tools.

- **Find Classes:** Starts the Find Classes UI that finds classes that can be tested by jtest!

Help

Commands in this menu provide additional information about the jtest! system.

- **Contents [F3]:** Opens the jtest! User's Guide.
- **License:** Lets you enter and view your jtest! license.
- **Environment:** Provides more information about the environment in which jtest! is running.
    - **Show CLASSPATH:** Displays the CLASSPATH jtest! uses when it tests a class.
    - **Show User:** Display the name of the current jtest! user.
    - **Show OS:** Displays the operating system that jtest! is currently running on.
    - **Show Java:** Displays the Java version being used to run the jtest! UIs.
- **Feedback:** Displays information about how to send ParaSoft feedback about jtest!.
- **About:** Displays the jtest! version number and logo.

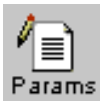
  Context-sensitive help is also available. To enable context-sensitive help, click the **Help** button in the toolbar, then move your cursor over the item that you would like additional information about.

Toolbar

The following commands are available in the jtest! Class Testing UI toolbar.

Note: Buttons with a small downward arrow in their top right-corner have additional commands available in a shortcut menu. To access the shortcut menu containing additional commands, right-click the button.

Button	Name	Action
	New Session	Starts a new session by clearing out any existing setup values or settings.
	Project Testing UI	Opens the Project Testing UI (used to test a set of classes).
	Start All Tests	Starts testing the class. Right-clicking this button displays the following commands in a shortcut menu: • **Start All Tests:** Start testing the class. • **Static Analysis Only:** Starts Static Analysis only. • **Dynamic Testing Only:** Starts Dynamic Analysis only

	Stop	Stops testing the class.
	View Report	Displays the Single Class Report for the current test. Right-clicking this button displays the following commands in a shortcut menu: • **View Report:** Displays the Summary Report of the current test. • **Print Report:** Sends the report directly to the printer.
	View Test Cases	Opens the View Test Cases window, which displays test cases that jtest! used for Dynamic Analysis.
	**Class Test Parameters**	Lets you view and edit the Class Test Parameters (parameters used for this class test).
	Global Test Parameters	Lets you view and edit the Global Test Parameters.

	View Class Source	Displays the source of the class currently under test. Right-clicking this button displays the following commands in a shortcut menu: • **View Class Source:** Displays the source of the current class. • **Edit Class Source:** Displays the source of the current class in an editor. • **Compile Class:** Compiles the source of the current class.
	Context Help	Enables context-sensitive help. After clicking this button, move your cursor over the area on the UI that you would like to learn more about, and, if that area has context-sensitive help, a help window will open.

Class Name Panel

This panel lets you specify what class you want jtest! to test.

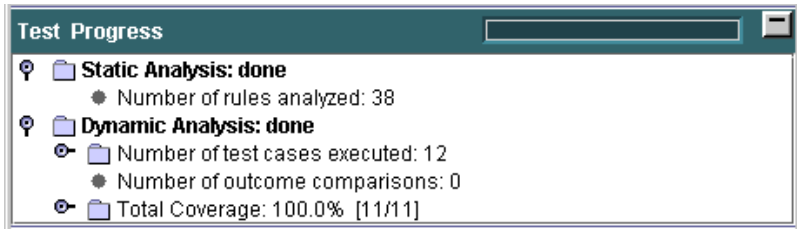
To browse for the class to be tested, click the **Browse** button, locate and choose the file you want to test in the file viewer, then choose **Open**.

Note: It is recommended that the class to be tested is selected using the **Browse** button. When a class is selected using the **Browse** button, the working directory is set to the root directory of the class' package.

To enter a class directly, enter the fully qualified name of the class to test in the **Class Name** field, without the .class extension.

Test Progress Panel

This panel displays the following test progress and coverage information:


Test Progress
Static Analysis: done
Number of rules analyzed: 38
Dynamic Analysis: done
Number of test cases executed: 12
Number of outcome comparisons: 0
Total Coverage: 100.0% [11/11]

- **Static Analysis:** Displays the progress of Static Analysis tests.
  - **Number of Rules Analyzed:** Displays number of static analysis rules analyzed.
- **Dynamic Analysis:** Displays the progress of Dynamic Analysis tests.
  - **Number of Test Cases Executed:** Displays the total number of test cases executed.
    - **Automatic:** Displays the number of automatically generated test cases executed.
    - **User Defined:** Displays number of user defined test cases executed.
  - **Number of outcome comparisons:** Number of class outcomes compared during black-box and regression testing.
  - **Total Coverage:** Displays the cumulative coverage jtest! achieved.
    jtest! performs data coverage for the generated input categories; this means that the parts of the class that have been covered are thoroughly tested with respect to those inputs. The coverage reported is relative to the classes that have been accessed for the paths jtest! has tried. If some part of the class is not covered, it means that jtest! has not yet found a path leading to those statements or no path leads to those statements. In class testing mode, approximately 50% of a class's code can be tested by jtest! Sometimes jtest! will be able to test 100% of the class, and sometimes it will test less than 50% of the class.

    - **Multi-condition branch:** Displays coverage achieved on multi-condition branches.
    - **Method:** Displays coverage achieved on methods.
    - **Constructor:** Displays coverage achieved on constructors.

> **Note:** Dynamic coverage is shown only for classes that dynamic analysis has been performed on (by default, dynamic analysis is only performed on the public classes. Static analysis is performed on all classes found (public and non-public).

While each type of analysis is being performed, a percentage indicating test progress is displayed to the right of the folder. When a test is complete, the word "done" will appear to the right of the folder.

Errors Found Panel

This panel displays information about the errors jtest! found and lets you perform numerous actions that provide more information about results and customize results.

Errors Found
[1] Static Analysis: done
[1] Uncaught Runtime Exceptions: done
Test Case Input
[2] Specification and Regression Errors: done

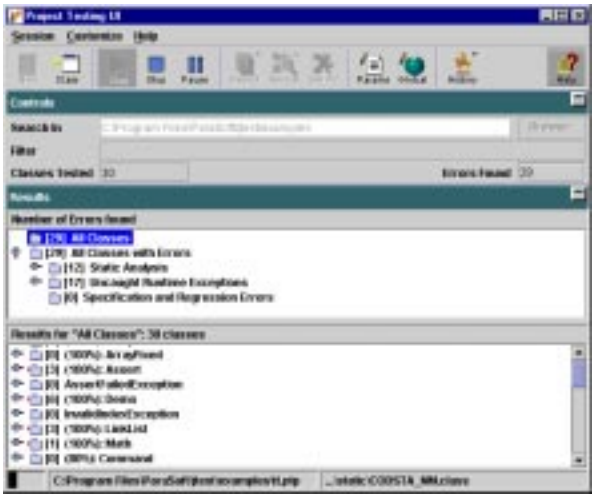
To learn more about this panel's branches and available options, see Understanding the Errors Found Panel and Understanding and Customizing Class Test Results.

Project Testing UI

The Project UI automatically tests sets of classes. Components of this UI are:

- Menu Bar
- Toolbar
- Project Controls Panel
- Results Panel

By default, the Class Testing UI opens when jtest! is started. To configure jtest! so the Project Testing UI appears when jtest! is started, choose **Starting UI** from the **Customize** menu in either UI, then choose **Project Testing UI**.



For information on testing a set of classes in the Project Testing UI, see Testing a Set of Classes.

Project Testing UI Menu Bar

File

Commands in this menu control test session functionality.

- **New Session:** Starts a new session by clearing out any existing setup values or settings.
- **Open:** Allows you to opens an existing test specification; opens a dialog box in which you can specify a test parameters file.
- **Open Recent:** Allows you to open parameters of a recent test. Contains a list of the most recently opened classes; choose a file name from the list to open the associated parameters file.
    - **Clear List:** Clears all items from this list.
- **Save:** Saves the current class test parameters into the test parameters file shown in the status bar.
- **Save As:** Saves the current class test parameters into a user-specified file.
- **Class Testing UI:** Opens the Class Testing UI (used to test a single class or view results for a single class).
- **Close UI:** Closes the Project Testing UI. If the Class Testing UI is not open, choosing this command will also close jtest!
- **Exit:** Closes jtest!

Preferences

Commands in the menu let you customize jtest! system settings.

- **Editor:** Opens a dialog box that allows you to determine what editor is invoked when you view report files and edit your source. If the editor command includes white-space, be sure to enclose the command in quotation marks. To represent the file parameter and the line number parameter, use the special tokens $FILE and $LINE.
- **Title Bar Background Color:** Determines the title bar's background color.
- **Starting UI:** Determines what UI opens by default when jtest! is started.
- **Context Help Font:** Determines the size and type of the font used to display context-sensitive help text.
- **Look and Feel:** Changes the look and feel of the jtest!'s UIs.
- **Tips:** Contains options that configure context-sensitive tips.
    - **Reactivate All:** Reactivates all context-sensitive tips.
    - **Deactivate All:** Turns off all content-sensitive tips.

- **Default Results Loading:** Determines what results are loaded when a .ptp file is opened.
    - **last:** Causes only results from the most recent test to be loaded when a .ptp file is opened.
    - **all:** Causes all previous results (including results from the most recent test run) to be loaded when a .ptp file is opened.
    - **none:** Prevents any results from being loaded when a .ptp file is opened.
- **Report Type:** Determines how jtest! reports are formatted.
    - **ASCII:** Prompts jtest! generate reports in ASCII format.
    - **HTML:** Prompts jtest! to generate reports in HTML format.

Help

Commands in this menu provide additional information about the jtest! system.

- **Contents [F3]:** Opens the jtest! User's Guide.
- **License:** Lets you enter and view your jtest! license.
- **Environment:** Provides more information about the environment in which jtest! is running.
    - **Show CLASSPATH:** Displays the CLASSPATH jtest! uses when it tests a class.
    - **Show User:** Display the name of the current jtest! user.
    - **Show OS:** Displays the operating system that jtest! is currently running on.
    - **Show Java:** Displays the Java version being used to run the jtest! UIs.
- **Feedback:** Displays information about how to send ParaSoft feedback about jtest!.
- **About:** Displays the jtest! version number and logo.

  Context-sensitive help is also available. To enable context-sensitive help, click the **Help** button in the toolbar, then move your cursor over the item that you would like additional information about.

Project Testing UI Toolbar

The following commands are available in the Project UI toolbar.

Note: Buttons with a small downward arrow in their top right-corner have additional commands available in a shortcut menu. To access the shortcut menu containing additional commands, right-click the button.

Button	Name	Action
	New Session	Starts a new session by clearing out any existing setup values or settings.
	Class Testing UI	Opens the Class Testing UI (used to test a single class or view results for a single class).
	Start finding and testing classes	Starts finding and testing classes, starting in the **Search In** parameter. *Note: jtest! will not test a previously tested test unless that class was modified since the last test.*
	Stop	Stops finding and testing classes.
	Pause	Temporarily stops finding and testing classes. Click this button again to resume testing. *Note: jtest! will finish testing the current class before pausing.*

	View Report	Displays a report of the current project test. The report contains only the classes and errors that are contained in the Results panel at the time this button is clicked. To limit the classes and errors contained in your report, display only the desired classes and errors in the Results panel before you click the **View Results** button. Right-clicking this button displays the following commands in a shortcut menu: • **View Report:** Displays the Project Report (contains project test parameters and details on all errors available in the Results panel). • **View Detail Report:** Displays the Detailed Project Report (contains project test parameters, class test parameters, and all information available in the Results panel). • **View Summary Report:** Displays the Summary Project Report (contains one line for each error available in the Results panel).

	View All Results	Displays all results. Right-clicking this button displays the following commands in a shortcut menu: • **View All Results:** Displays all results. • **View Results From Last Run:** Displays only results from the last run.
	Delete All	Removes all the results in the lower Results window.
	Project Test Parameters	Lets you view and edit the Project Test Parameters which are used for the current Project test.
	Global Test Parameters	Lets you view and edit the Global Test Parameters.
	Test History	Displays a record of all the runs for this Project test, or all Project tests. Right-clicking this button displays the following commands in a shortcut menu: • **Test History:** Displays a record of all runs of the current test. • **Global History:** Displays a record of all Project tests.

	Context Help	Enables context-sensitive help. After clicking this button, move your cursor over the area on the UI that you would like to learn more about, and, if that area has context-sensitive help, a help window will open.

Controls Panel

This panel lets you specify the fundamental parameters used during a project test, and reports basic data about a project test.

You can enter two parameters in this panel:

- **Search In:** Specifies where jtest! should start searching for classes to test. The parameter can be a directory, a jar file, or a zip file.
  If the parameter is a directory, jtest! will recursively traverse the path's subdirectories, zip files, and jar files, searching for and testing any classes it finds.
  If the parameter is a jar or zip file, jtest! will open the file and search it for classes in which to find errors.
  To browse for the directory, jar file, or zip file that you want jtest! to start searching and testing, click the **Browse** button, locate and select the desired directory, jar file, or zip file, in the file viewer, then click **Open**.

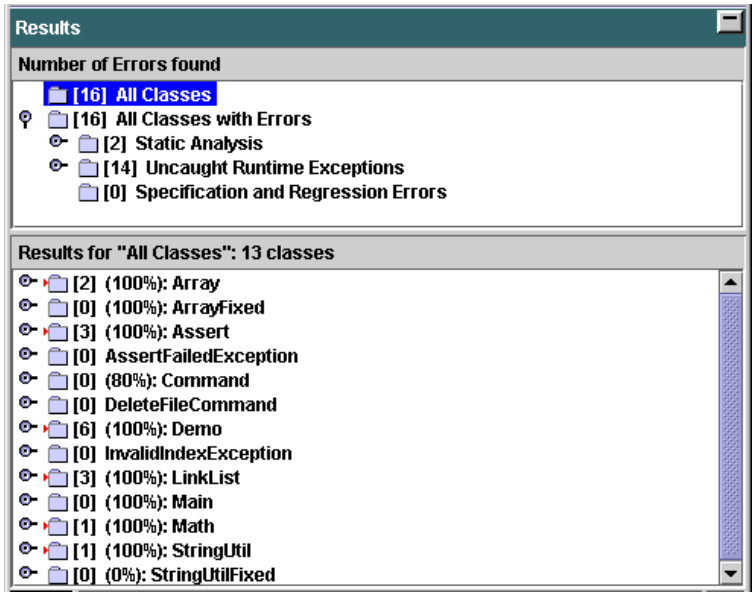
- **Filter-in:** Tells jtest! to only find and test classes that match the given expression. Use the * (star) character to match zero or more characters. For example, if you want jtest! to only look for classes in the DB package, enter the following parameter in this field
  `DB.*`
  When this field is left empty, all classes found will be tested.

Two test result parameters are displayed in this panel:

- **Classes Tested:** Displays the number of classes tested by jtest!.
- **Errors Found:** Displays the number of errors found by jtest!.

Project Testing UI Results Panel

This panel displays information about the errors jtest! found during a project test. It also lets you perform numerous actions that provide more information about results and customize results.



To learn more about this panel's branches and available options, see Understanding the Results Panel and Understanding and Customizing Project Test Results.

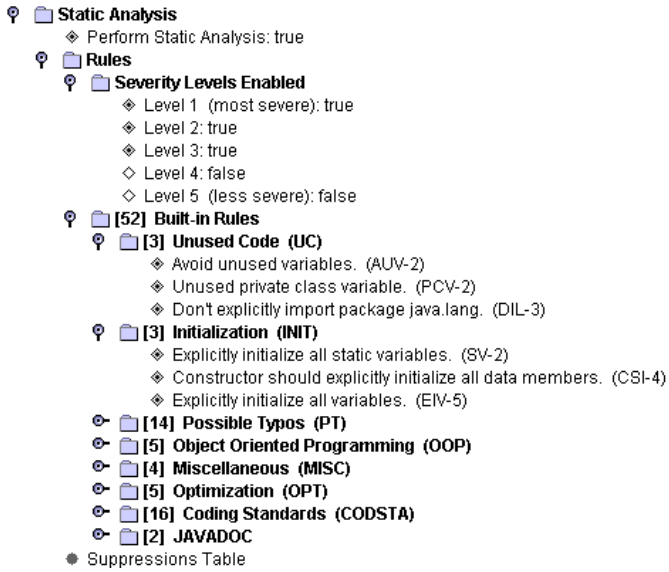
Global Test Parameters

This window lets you view and edit the global parameters used throughout jtest!. To open this window, click the **Global** button in either the Class Testing UI or Project Testing UI.

Descriptions of global test parameter tree branches are divided into three categories:

- Static Analysis Parameters
- Dynamic Analysis Parameters
- Common Parameters

Global Test Parameters - Static Analysis



Static Analysis

Contains parameters that control Static Analysis.

Perform Static Analysis

Flag that controls whether or not Static Analysis is performed every time a class is tested.

Note: This flag appears in all parameter levels.

Rules

Contains rules that are applied when performing the Static Analysis. jtest! reports violations of those rules.

Severity Levels Enabled

Contains severity level flags. Each rule has a severity level associated with it. A rule is applied if both the rule and its severity level are enabled. This

branch controls which severity levels are enabled.

Level 1-5

Flags that control whether or not the rules of a particular severity level are applied.

Built-in Rules

Contains built-in rules provided with jtest! Specific rules can be enabled or disabled using the flag associated with each particular rule (listed by category).

Unused Code

Contains rules that check for unused code.

Initialization

Contains rules that enforce the explicit initialization of the variables.

Possible Typos

Contains rules that check for possible typos in the code (i.e., the code compiles, but the programmer inadvertently entered the incorrect code).

Coding Standards

Contains rules that enforce common naming conventions and programming practices.

Object-Oriented Programming

Contains rules that enforce good Object-Oriented Programming practices.

Miscellaneous

Contains miscellaneous rules.

Optimization

Contains rules that check for non-optimal constructs.

JAVADOC

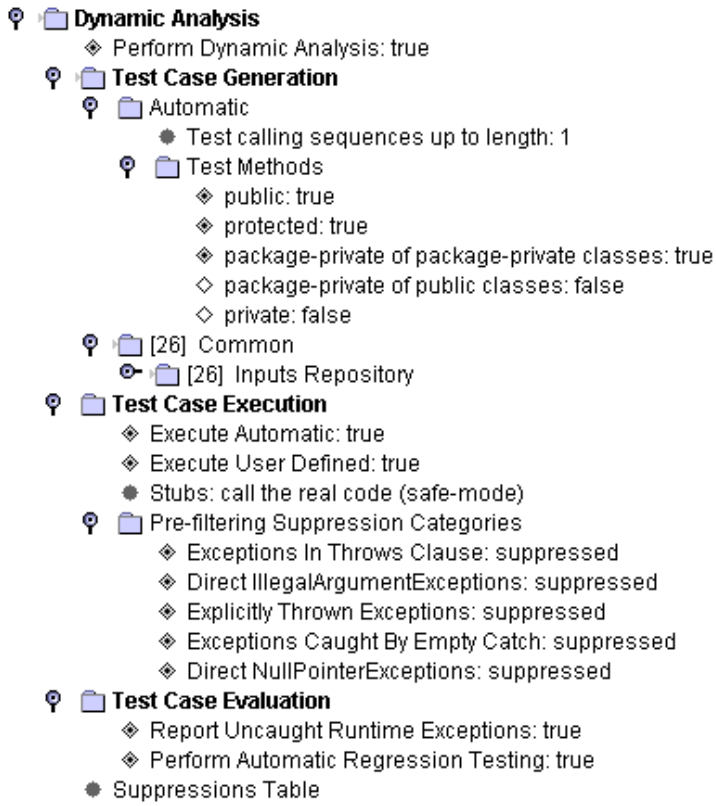
Contains rules related to Javadoc comments.

Suppressions Table

Double-clicking this leaf invokes the Static Analysis Suppression Table,

which lets you suppress static analysis violations.

Global Test Parameters - Dynamic Analysis



Dynamic Analysis

Contains parameters that control Dynamic Analysis.

Perform Dynamic Analysis

Flag that controls whether or not Dynamic Analysis is performed every time a class is tested.

Note: This flag appears in all parameter levels.

Test Case Generation

Contains parameters that control the generation of the test cases.

Automatic

Contains parameters that control the generation of automatic test cases.

Test calling sequences up to length

By default, jtest! tests each method by calling it independently and generating arguments to it. That is, jtest! basically tries calling sequences of length 1.

This option can be used to tell jtest! to try calling sequences longer than 1. If a calling sequence of length N is specified, jtest! will first try all calling sequences of length 1, then all calling sequences of length 2, and so on.

Note: jtest! will attempt to show errors with the shortest calling sequences that shows them. Most errors should show up with calling sequences of length 1 or 2.

Test Methods

Contains flags that control which methods can be called directly in the calling sequence generated by jtest!

jtest! will only call directly the methods whose accessibility is selected here.

Note that the methods that are not called directly are still tested indirectly, through calls to the methods that are called directly.

public

Flag that controls if jtest! tests all of the class's public methods.

protected

Flag that controls if jtest! tests all of the class's protected methods.

package-private of package-private classes

Flag that controls if jtest! tests all package-private methods in package-private classes.

A package-private method is a method without any accessibility qualifier (e.g., public, protected, or private). package-private methods are only accessible by classes within the same package as the method.

A package-private class is a class without the “public” accessibility qualifier. package-private classes are only accessible by other classes within the same package as the method.

package-private of public classes

Flag that controls if jtest! tests all package-private all package-private methods in public classes.

A package-private method is a method without any accessibility qualifier (e.g., public, protected, or private). package-private classes are only accessible by classes within the same package as the method.

private

Flag that controls if private methods are directly called.

Common

Contains parameters shared by both the automatic and user-defined test case generation.

Inputs repository

Once a value is added to the repository, the input can be added to any input node for a class by choosing the associated **Add From Global Repository** shortcut menu item for that node. To add input values to this repository, right-click this node and use the following commands:

- **Add Constant:** Adds an entry that can be expressed as a single expression (e.g., "int ONE= 1") to this Inputs Repository.
- **Add Method:** Adds a complicated entry (an entry is considered complicated if calculation of the input requires complicated calculations that cannot be represented by a single expression) to this Inputs Repository.
- **Set Factory Defaults:** Resets all inputs repository values to their factory default values.

For more information on the inputs repository, see Adding Inputs.

Test Case Execution

Contains parameters that control the execution of test cases.

Execute Automatic

Flag that controls whether or not automatic test cases are executed every time a class is tested.

Note: This flag appears in all parameter levels.

Execute User-Defined

Flag that controls whether or not the user-defined test cases are executes every time a class is tested.

Note: This flag appears in all parameter levels.

Stubs

This node specifies what jtest! should do when the class under test references methods external to the class. Available values are:

- **Call the real code (safe-mode):** Calls the actual external.

In safe mode, jtest! will not call natives other than the ones needed to do basic input.

When jtest! finds a native, it will stop the current path of execution and will try another path.

This prevents jtest! from possibly corrupting the computer resources (i.e. by overwriting files, or starting/using connections to sockets or databases).

Note: Future releases of jtest! will provide more options for this parameter.

Pre-filtering Suppression Categories

Contains suppression categories that can be applied when the test cases are executed.

Exceptions in Throws Clause

If selected, jtest! will not report exceptions occurring in methods that are declared with the exception's type in the throws clause of the method.

Note: This flag appears in all parameter levels.

DirectIllegalArgumentExceptions

If selected, jtest! will not report Illegal Argument Exceptions that are thrown

directly by a throw statement.

Note: This flag appears in all parameter levels.

Explicitly Thrown Exceptions

If selected, jtest! will not report exceptions that are explicitly thrown by user code with a throw statement.

Note: This flag appears in all parameter levels.

Exceptions Caught By Empty Catch

If selected, jtest! will not report exceptions caught by an empty catch block.

Note: This flag appears in all parameter levels.

DirectNullPointerExceptions

If selected, jtest! will not report exceptions that can occur because a null object is passed to a method which subsequently dereferences the object, thus causing the NullPointerException.

Note: This flag appears in all parameter levels.

Test Case Evaluation

Contains parameters that control the evaluation of the test cases.

Report Uncaught Runtime Exceptions

Flag that controls whether or not jtest! will report Uncaught Runtime Exceptions that occur in the tested class.

Note: This flag appears in all parameter levels.

Perform Automatic Regression Testing

Flag that controls whether or not jtest! will perform Automatic Regression Testing for the tested class.

Note: This flag appears in all parameter levels.

Suppressions Table

Double-clicking this leaf invokes the Dynamic Analysis Suppression Table, which lets you suppress dynamic analysis violations.

Global Test Parameters - Common Parameters

Common Parameters

Contains parameters shared by both Static and Dynamic Analysis.

Directories

Contains parameters related to directories.

Working directory

Determines the directory that is used as the current working directory when testing a class.

For example, this directory will be used as "." on the CLASSPATH.

This parameter appears in most parameter levels (Global, Project, and Class). When testing a class, jtest! uses the value in the Class Test Parameters. If this parameter is not set, the value of the current parent parameter is used.

Tokens of the form $NAME are replaced by the value of the environment variable NAME. If the environment variable PARAMS_DIR is not set, the token $PARAMS_DIR is replaced by the directory where the parameter's file resides.

In any case, the actual value that will be used is shown in parentheses.

Results

Determines where the test results will be stored.

The token $DEFAULT" receives special treatment in project tests; it is replaced by a path relative to the location of the project test parameters (.ptp) file.

This parameter appears in most parameter levels (Global, Project, and Class). When testing a class, jtest! uses the value in the Class Test Parameters. If this parameter is not set, the value of the current parent parameter is used.

Tokens of the form $NAME are replaced by the value of the environment variable NAME. If the environment variable PARAMS_DIR is not set, the token $PARAMS_DIR is replaced by the directory where the parameter's file resides.

In any case, the actual value that will be used is shown in parentheses.

javac/javac-like Parameters

Contains parameters equivalent to parameters used in java or javac.

-classpath

Overrides the CLASSPATH environment variable with the list of entries specified here (an entry is a directory, zip file, or jar file).

This option is equivalent to the java interpreter's -classpath flag.

The token $PARENT receives special treatment and is replaced by the parent parameter value.

This parameter appears in most parameter levels (Global, Project, and Class). When testing a class, jtest! uses the value in the Class Test Parameters. If this parameter is not set, the value of the current parent parameter is used.

Tokens of the form $NAME are replaced by the value of the environment variable NAME. If the environment variable PARAMS_DIR is not set, the token $PARAMS_DIR is replaced by the directory where the parameter's file resides.

In any case, the actual value that will be used is shown in parentheses.

-cp

Prepends the CLASSPATH environment variable with the list of entries spec-

ified here (an entry is a directory, zip file, or jar file).

This option is equivalent to the java interpreter's -cp flag.

The token $PARENT receives special treatment and is replaced by the parent parameter value.

This parameter appears in most parameter levels (Global, Project, and Class). When testing a class, jtest! uses the value in the Class Test Parameters. If this parameter is not set, the value of the current parent parameter is used.

Tokens of the form $NAME are replaced by the value of the environment variable NAME. If the environment variable PARAMS_DIR is not set, the token $PARAMS_DIR is replaced by the directory where the parameter's file resides.

In any case, the actual value that will be used is shown in parentheses.

System Properties

Defines system properties. This parameter is equivalent to the -D flag of the java interpreter and is used to define properties to the class being tested.

System properties are defined by naming the property and assigning a value to the property. Use a space to separate properties if multiple properties are defined.

Example: `property.one=On PROPERTY_TWO=d:/temp`

This parameter appears in most parameter levels (Global, Project, and Class). When testing a class, jtest! uses the value in the Class Test Parameters. If this parameter is not set, the value of the current parent parameter is used.

Tokens of the form $NAME are replaced by the value of the environment variable NAME. If the environment variable PARAMS_DIR is not set, the token $PARAMS_DIR is replaced by the directory where the parameter's file resides.

In any case, the actual value that will be used is shown in parentheses.

Source Path

Determines where jtest! looks for the source of a class.

Path to JDK 1.2 Installation

Specifies the path to the JDK 1.2 installation.

Class Test Parameters

This window lets you view and edit parameters that are specific to a certain class.

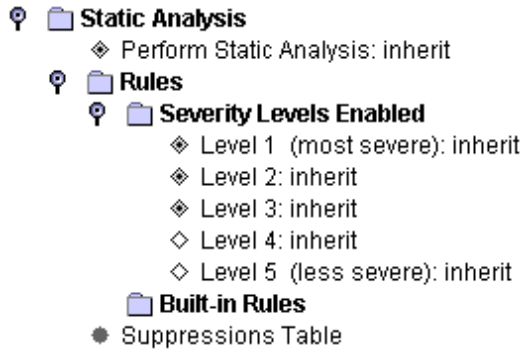
In the Class Testing UI, you can open this window by clicking the **Params** button.

You can also open this window from the Project Testing UI.

Descriptions of class test parameters tree branches are divided into three categories:

- Static Analysis Parameters
- Dynamic Analysis Parameters
- Common Parameters

Class Test Parameters - Static Analysis



Static Analysis

See description in Global Test Parameters.

Perform Static Analysis

See description in Global Test Parameters.

Rules

See description in Global Test Parameters.

Severity Levels

See description in Global Test Parameters.

Level 1-5

See description in Global Test Parameters.

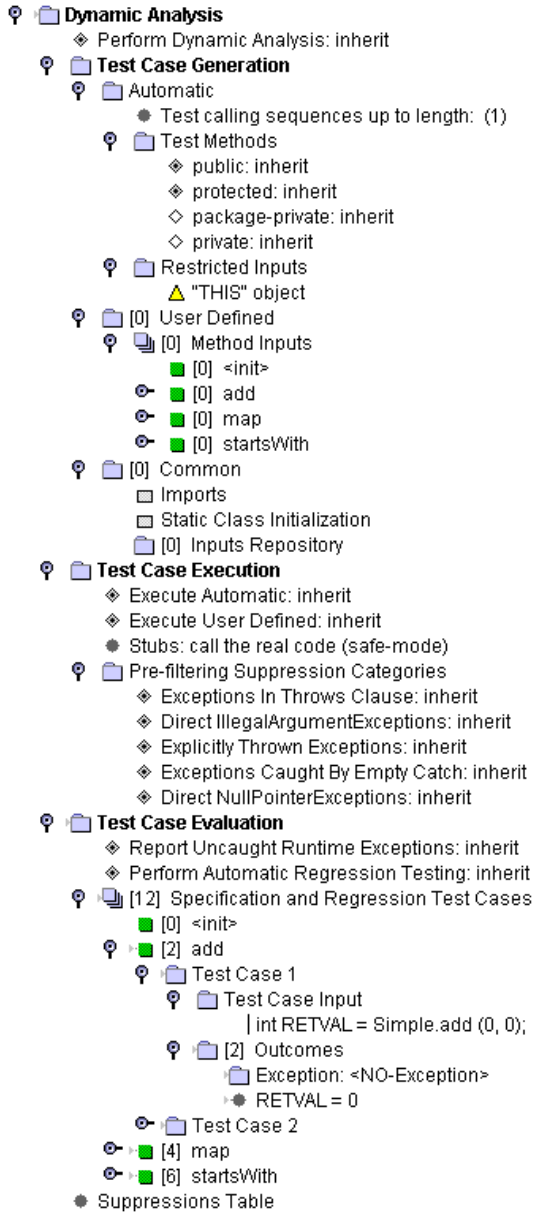
Built-in Rules

Built in rules need to be enabled/disabled from the the Global Test Parameters window.

Suppressions Table

See description in Global Test Parameters.

Class Test Parameters - Dynamic Analysis



Dynamic Analysis

See description in Global Test Parameters.

Perform Dynamic Analysis

See description in Global Test Parameters.

Test Case Generation

See description in Global Test Parameters.

Automatic

See description in Global Test Parameters.

Test calling sequences up to length

See description in Global Test Parameters.

Test Methods

See description in Global Test Parameters.

public

Flag that controls if public methods are directly called.

protected

Flag that controls if protected methods are directly called.

package-private

Flag that controls if package-private methods are directly called.

private

Flag that controls if private methods are directly called.

Restricted Inputs

By default, jtest! will try to generate any input for the methods of the class. Use these nodes to restrict the inputs jtest! will generate.

"THIS" object

Specifies what value jtest! will use by default when testing instance methods of the given class. Right-clicking a method displays a shortcut menu that allows you to set restricted input, add from local repository, or add from global repository.

- **Set Restricted Input:** Lets you add a simple input value.
- **Add From Local Repository:** Contains menu items associated with the values available in the local repositories.
  Choose an input's menu item to add that input to the node.
  You can add inputs to the local repository in **Class Test Parameters> Dynamic Analysis> Test Case Generation> Common> Inputs Repository**.
- **Add From Global Repository:** Contains menu items associated with the values available in the global repository.
  Choose a menu item to add the input to the node.
  You can add inputs to the global repository in **Global Test Parameters> Dynamic Analysis> Test Case Generation> Common> Inputs Repository**.

User defined

Contains parameters that control the generations of the user defined test cases. [n]= number of test cases defined.

Method Inputs

Contains nodes that can be used to specify the set of inputs with which you want jtest! to test the class. [n]= number of test cases.

[Method name]

Use these nodes to specify the inputs to be used for the named method. [n]= number of test cases for this method.

[Argument name]

Use the associated shortcut menu to add values to this argument. [n]= number of inputs for this argument.

Note: Shortcut menu commands available include:

- **Add Input Value:** Lets you add a simple input value.
- **Add From Local Repository:** Contains menu items associated with the values available in the local repositories.
  Choose an input's menu item to add that input to the node.
  You can add inputs to the local repository in **Class Test Parameters> Dynamic Analysis> Test Case Generation> Common> Inputs Repository**.
- **Add From Global Repository:** Contains menu items associated with the values available in the global repository.
  Choose a menu item to add the input to the node.

You can add inputs to the global repository in **Global Test Parameters> Dynamic Analysis> Test Case Generation> Common> Inputs Repository**.

- **Delete All Inputs:** Removes all existing inputs.

Common

Contains parameters shared by both the automatic and user-defined test case generation

Imports

Contains imports shared by all the code used in the specification. See Specifying Imports for more information.

Static Class Initialization

The code associated with this node is executed before any test case is executed, and can be used to setup and initialize the class if needed. See Setting an Object to a Certain State for more information.

Inputs Repository

Once a value is added to the repository, the input can be added to any input node for this class by choosing the associated **Add From Local Repository** shortcut menu item for that node.

To add input values to this repository, right-click this node and use the following commands:

- **Add Constant:** Adds an entry that can be expressed as a single expression (e.g., "int ONE= 1") to this Inputs Repository.
- **Add Method:** Adds a complicated entry (an entry is considered complicated if calculation of the input requires complicated calculations that cannot be represented by a single expression) to this Inputs Repository.

For more information on the inputs repository, see Adding Inputs.

Test Case Execution

See description in Global Test Parameters.

Execute Automatic

See description in Global Test Parameters.

Execute User-Defined

See description in Global Test Parameters.

Stubs

See description in Global Test Parameters.

Pre-filtering Suppression Categories

See description in Global Test Parameters.

Exceptions in Throws Clause

See description in Global Test Parameters.

DirectIllegalArgumentExceptions

See description in Global Test Parameters.

Explicitly Thrown Exceptions

See description in Global Test Parameters.

Exceptions Caught By Empty Catch

See description in Global Test Parameters.

DirectNullPointerExceptions

See description in Global Test Parameters.

Test Case Evaluation

Contains parameters that control the evaluation of the test cases.

The color of the arrow to the left of a leaf has the following meaning:

- **green:** the outcome is correct, or has been validated as correct by the user.
- **red:** the outcome is incorrect or has been validated as incorrect by the user.
- **gray:** the outcome status is unknown.
- **no arrow:** the user has specified to ignore this outcome.

If the **Perform Automatic Regression Testing** flag is selected, jtest! will assume that gray outcomes are correct and will report an error if the outcome changes.

In the View Test Cases window, the outcome is marked as incorrect if it is different from the one in the Specification and Regression Test Cases. When

more than one test case outcome differs, only one of them is marked as an error and only that one is reported as an error in the Errors Found panel.

Test cases can be deleted by right-clicking a test case leaf and choosing **Delete** from the shortcut menu.

Right-click a specific outcome to indicate whether an outcome is correct or incorrect, if you want jtest! to ignore this outcome, or to tell jtest! that the correct outcome is unknown. You can also edit the value for an outcome by right-clicking, choosing **Edit** from the shortcut menu, and modifying the value in the dialog box that opens.

To indicate that all outcomes for a test case are correct, right-click the appropriate **Outcomes** leaf and choose **Set All Correct** from the shortcut menu.

Report Uncaught Runtime Exceptions

See description in Global Test Parameters.

Perform Automatic Regression Testing

See description in Global Test Parameters.

Specification and Regression Test Cases

These test cases are used as reference cases to do regression and black-box testing.

When tests are run, the outcomes for the run are compared with these outcomes. If a discrepancy exists, an error is reported.

jtest! automatically adds the automatic test cases that increase coverage to this list. The user-defined test cases are always added.

A black-box review can be performed by going over the test cases and checking if the class outcomes follow the class specification. Shortcut menus allow you to specify whether outcomes are correct or incorrect. (n=number of test cases)

[method name]

Contains test cases for this method.

Test Cases

Contains all the information for a test case.

Test Case Input

Input that defines the test case.

The input for automatic test cases is the calling sequence.

Outcomes

Outcomes for this test case. Verify if the outcomes are correct or incorrect according to the class specification and set their state using the shortcut menus.

When the outcome is an object, jtest! automatically chooses the toString method to show its state.

If a method called jtestInspector is defined for the object's class, jtest! will only use the return value of this method to show the object state.

If no toString or jtestInspector methods are defined, jtest! will heuristically choose some public instance methods for that object to show its state.

If the method under test is a static method, jtest! will heuristically choose public static methods to show the class state. If the methods jtest! chose are not enough, declare a static method called sjtestInspector for the class. jtest! will use the return value of this method to show the object class.

[n]= number of outcomes for this test case.

Exceptions

Indicates whether an exception occurred, and, if so, what type of exception occurred.

Suppressions Table

See description in Global Test Parameters

Class Test Parameters - Common Parameters

Common Parameters

See description in Global Test Parameters.

Directories

See description in Global Test Parameters.

Working Directory

See description in Global Test Parameters.

Results

See description in Global Test Parameters.

javac/javac-like Parameters

See description in Global Test Parameters.

-classpath

See description in Global Test Parameters.

-cp

See description in Global Test Parameters.

System Properties

See description in Global Test Parameters.

Project Test Parameters

This window lets you view and edit parameters that apply to the current project test. To open this window, click the **Params** button in the Project Testing UI tool-bar.

Descriptions of project test parameter tree branches are divided into three categories:

- Static Analysis Parameters
- Static Analysis Parameters
- Common Parameters, Search Parameters, Classes in Project

Project Test Parameters - Static Analysis Parameters

Static Analysis

See description in Global Test Parameters.

Perform Static Analysis

See description in Global Test Parameters.

Rules

See description in Global Test Parameters.

Severity Levels Enabled

See description in Global Test Parameters.

Level 1-5

See description in Global Test Parameters.

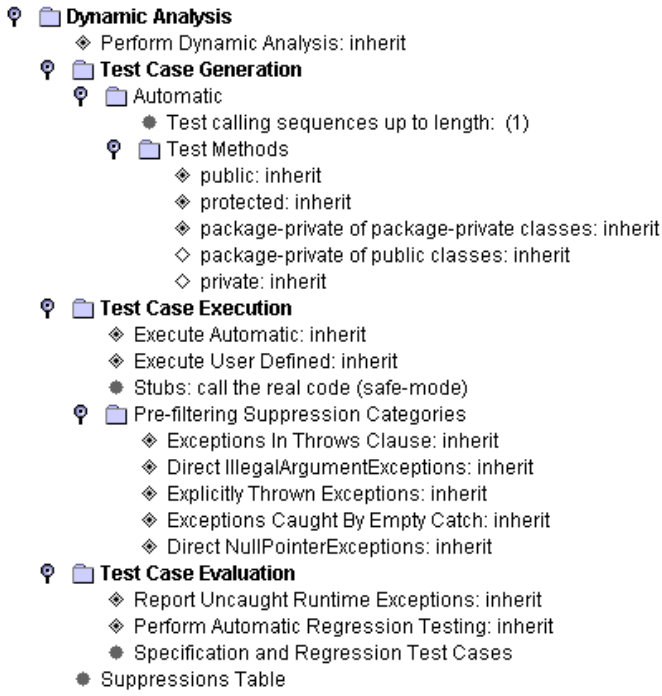
Built-in Rules

Built in rules need to be enabled/disabled from the Global Test Parameters window.

Suppressions Table

See description in Global Test Parameters.

Project Test Parameters - Dynamic Analysis Parameters



Dynamic Analysis

See description in Global Test Parameters.

Perform Dynamic Analysis

See description in Global Test Parameters.

Test Case Generation

See description in Global Test Parameters.

Automatic

See description in Global Test Parameters.

Test calling sequences up to length

See description in Global Test Parameters.

Test Methods

See description in Global Test Parameters.

public

See description in Global Test Parameters.

protected

See description in Global Test Parameters.

package-private of package-private classes

See description in Global Test Parameters.

package-private of public classes

See description in Global Test Parameters.

private

See description in Global Test Parameters.

Test Case Execution

See description in Global Test Parameters.

Execute Automatic

See description in Global Test Parameters.

Execute User-Defined

See description in Global Test Parameters.

Stubs

See description in Global Test Parameters.

Pre-filtering Suppression Categories

See description in Global Test Parameters.

Exceptions in Throws Clause

See description in Global Test Parameters.

DirectIllegalArgumentExceptions

See description in Global Test Parameters.

Explicitly Thrown Exceptions

See description in Global Test Parameters.

Exceptions Caught By Empty Catch

See description in Global Test Parameters.

DirectNullPointerExceptions

See description in Global Test Parameters.

Test Case Evaluation

See description in Global Test Parameters.

Report Uncaught Runtime Exceptions

See description in Global Test Parameters.

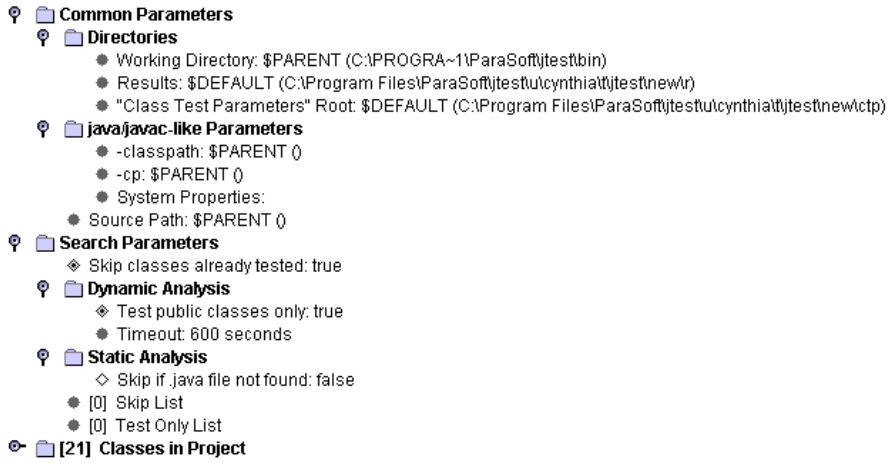
Perform Automatic Regression Testing

See description in Global Test Parameters.

Suppressions Table

See description in Global Test Parameters.

Project Test Parameters - Common Parameters, Search Parameters, Classes in Project



Common Parameters

See description in Global Test Parameters.

Directories

See description in Global Test Parameters.

Working directory

See description in Global Test Parameters.

Results

See description in Global Test Parameters.

Class Test Parameters Root

When you test a project, the Project Testing UI automatically creates the class test parameters for the individual classes found. This parameter deter-

mines what directory the class test parameter (.ctp) files are stored in.

The string $DEFAULT" receives special treatment; it is replaced by a path relative to the location of the project test parameters (.ptp) file.

This parameter appears in most parameter levels.

javac/javac-like Parameters

See description in Global Test Parameters.

-classpath

See description in Global Test Parameters.

-cp

See description in Global Test Parameters.

System Properties

See description in Global Test Parameters.

Search Parameters

Contains parameters that control how jtest! searches for classes.

Skip classes already tested

If selected, jtest! will not retest a class if results for that class already exists and the class didn't change since the previous results were calculated. jtest! determines whether or not a class has changed by checking that both the .class file and the .java file contents have not changed. Timestamps are not considered.

Dynamic Analysis

Contains parameters that control how jtest! searches for classes for dynamic analysis.

Test public classes only

If selected, jtest! will only perform dynamic analysis on public classes.

Note that the non-public classes will be tested indirectly when called from the public classes.

Timeout

Specifies the maximum amount of time jtest! will spend testing any one class in the project.

Static Analysis

Contains parameters that control how jtest! searches for classes for static analysis.

Skip if .java file not found

If selected, jtest! will only perform static analysis on classes for which it finds a .java file.

Skip list

Opens a dialog box which lets you enter the names of specific classes in your project that you do not want tested.

Test Only list

Opens a dialog box which lets you enter the names of specific classes in your project that you want tested.

Classes in Project

Contains a list of all classes in the project. Also allows you to suspend and resume the finder's search for classes, and delete all individual class test parameters.

- To suspend the finder from searching for all classes in the project, right-click this node and choose **Suspend Finder** from the short-cut menu.
- To prompt the finder to resume finding classes in this project, right-click this node and choose **Resume Finder** from the shortcut menu.
- To delete all individual Class Test Parameters, right-click this node and choose **Delete all individual Class Test Parameters** from the shortcut menu.

[Class Name]

Allows you to edit and reset the named class's class test parameters, and open the named class in the Class Testing UI.

- To edit class test parameters, right-click this node and choose **Edit Class Test Parameters** from the shortcut menu.

- To reset all class test parameters to their default value, right-click this node and choose **Reset Class Test Parameters** from the shortcut menu.
- To load this class in the class testing UI (where you can focus on results for this class), right-click this node and choose **Load Test in Class Testing UI** from the shortcut menu.

Find Classes UI

The Find Classes UI searches for classes that can be tested by jtest!, then allows you to easily set up a test for any found class. This UI can be opened by choosing **Find Classes UI** from the **Tool** menu. To find classes, tell the Find Classes where to start finding classes (using the **Browse** button, or by entering the path in the **Search In** field), then click the **Start** button.

Find Classes Toolbar

The following commands are available in the Find Class UI toolbar:

Button	Name	Action
	Find	Starts finding classes, starting in the **Search In** parameter.
	Stop	Stops finding classes.
	Pause	Temporarily stops finding classes.

Find Classes Panel

- **Only find public classes:** If checked, jtest! will only find public classes when the **Find** button is clicked.

  Search In: Specifies where jtest! should start searching for classes to test. The parameter can be a directory, a .class file, a jar file, or a zip file.

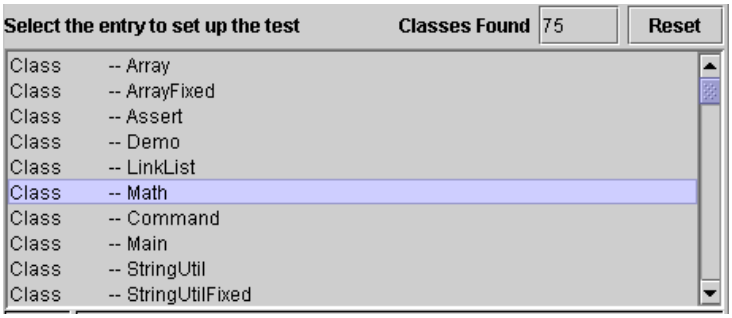
If the parameter is a directory, jtest! will recursively traverse the path's subdirectories, zip files, and jar files when it searches for file to test.
If the parameter is a jar or zip file, jtest! will open the file and search it for classes in which to find errors.
To browse for the directory, jar file, or zip file that you want jtest! to start searching, click the **Browse** button, locate and select the desired directory, jar file, or zip file, in the file viewer, then click **Open**.

- **Filter-in:** Tells jtest! to only find classes that match the given expression. Use the * (star) character to match zero or more characters.
  For example,if you want jtest! to only look for classes in the DB package, enter the following parameter in this field:
  `DB.*`
  When this field is left empty, jtest! will look for all classes.

- **Classes Found:** The number of classes found.

- **Reset:** Clears the lower panel.

The lower panel lists the classes that were located by the Find Classes.



To load a class found into the Class Testing UI (for testing), double-click the name of the class. The class will then be loaded in the Class Testing UI, and can be tested by clicking the **Start** button in the Class Testing UI.

Status Bar

The status bar displays the current search path.

Built-in Static Analysis Rules

Category	Rule	Description
Unused Code (UC)	UC_AUV-2	Avoid unused variables
	UC_PCV-2	Unused private class variable
	UC_DIL-3	Don't explicitly import package java.lang
Initialization (INIT)	INIT_SV-2	Explicitly initialize all static variables
	INIT_CSI-4	Constructor should explicitly initialize all data members
	INIT_EIV-5	Explicitly initialize all variables
Possible Typos (PT)	PT_SBC-1	Switch statement with bad case
	PT_ASI-4	Avoid assignment within an if condition
	PT_UEI-1	Use "equals" instead of "=="
	PT_FEB-1	Avoid for-statements with empty bodies
	PT_IEB_1	Avoid if-statements with empty bodies
	PT_ADE-2	Avoid dangling else statements
	PT_CLP-2	Don't cast primitive datatypes to lower precision
	PT_TLS-2	Don't use text labels in switch statements
	PT_AUO-2	Avoid using an object to access a class "static" variable or method

	PT_PDS-3	Provide "default" for each switch statement
	PT_NEA-3	Do not use embedded assignment
	PT_DCF-3	Don't compare floating point type
	PT_MPC-4	Avoid using method parameter names that conflict with class member names
	PT_DCP-4	Don't confuse + operator for String concatenation
Object-Oriented Programming (OOP)	OOP_AHF-1	Avoid hiding inherited static member functions
	OOP_AHV-2	Avoid hiding inherited instance variables
	OOP_APV-3	Avoid public and package instance variables
	OOP_LPF-4	List all public and package methods/data first
	OOP_IIN-5	Implement interfaces non-trivially or abstract
Miscellaneous (MISC)	MISC_HMV-2	Avoid hiding member variables in member functions
	MISC_PIF-3	Provide incremental in for-statement or use while statement
	MISC_PCF-3	Provide condition in for-statement
	MISC_EFB-5	Each for statement should have a block
Optimization (OPT)	OPT_CS-1	Close stream in finally block

	OPT_AAS-3	Use abbreviated assignment operator
	OPT_USC-3	Use "String" instead of "StringBuffer" for constant strings
	OPT_USB-4	Use "StringBuffer" instead of "String"
	OPT_DUN-4	Don't use the negation operator frequently
Coding Standards (CODSTA)	CODSTA_NCL-3	Enforce name format of classes
	CODSTA_NIF-3	Enforce name format of interfaces
	CODSTA_NE-3	Enforce name format of exceptions
	CODSTA_NM-3	Enforce name format of methods
	CODSTA_NCV-3	Enforce name format of class variables
	CODSTA_NCM-3	Enforce name format of class methods
	CODSTA_NMP-3	Enforce name format of method parameters
	CODSTA_NIV-3	Enforce name format of instance variables
	CODSTA_NLV-3	Enforce name format of local variables
	CODSTA_OGM-3	Order group members with the same name together
	CODSTA_USV-3	Use uppercase letters for final static variables
	CODSTA_PML-4	Put the main function last
	CODSTA_NTX-4	Never throw Class Exception directly

	CODSTA_NCE-4	Never use Exception or Runt-imeException in catch
	CODSTA_NTE-4	Never throw an "Error" directly
	CODSTA_CVN-5	Use conventional variable names
JAVADOC	JAVADOC_BT-2	Bad tag in javadoc comment
	JAVADOC_DC-5	Distinguish between javadoc and ordinary comments

Avoid unused variables (UC_AUV_2)

Description

jtest! can detect unused local variables in your methods.

Example

```
class AUV {
    void method () {
        int i = 4;
    }
}
```

Diagnosis

Unused variable "i" in method "test"

Repair

An unused variable possibly indicates a logical flaw in the corresponding method; in this case, the method needs to be rewritten to take the unused variable into account.

Unused private class variable (UC_PCV_2)

Description

jtest! can detect unused private class variables.

Example

```
public class UC_PCV {
    private static int i;
}
```

Diagnosis

Unused private class variable "i"

Repair

An unused variable possibly indicates a logical flaw in the corresponding class; in this case, the class needs to be rewritten to take the unused variable into account.

Don't explicitly import package java.lang (UC_DIL_3)

Description

jtest! can detect when the package java.lang is imported. It is not necessary to import this package as it is imported implicitly.

Example

```
import java.lang.*;

public class UC_DIL {
    public int method () {
        return 0;
    }
}
```

Diagnosis

Don't explicitly import package java.lang.

Repair

Remove import statement for "import java.lang.*"

Explicitly initialize all static variables (INIT_SV_2)

Description

jtest! can detect uninitialized static variables in your methods.

Example

```
public class INIT_SV {
    INIT_SV () {
        L = 5; K = 20;
    }

    static private int K, L;

    static {
        K = 10;
    }
}
```

Diagnosis

Static variable "L" not initialized by static initializer.

Repair

These errors can be corrected by initializing the appropriate static variables.

Constructor should explicitly initialize all data members (INIT_CSI_4)

Description

jtest! can detect constructors that do not initialize all class variables.

Example

```
public class INIT_CSI {

    INIT_CSI () {
        this (12);
        k = 0;
    }

    INIT_CSI () {
        j = val;
    }

    private int i = 5, j, k;
}
```

Diagnosis

Uninitialized variable "j" on line 12.

Uninitialized variable "k" on line 12.

Repair

In some cases, these errors can be corrected by initializing the appropriate variable. In other cases, these variables will have been assigned values from other uninitialized variables that must be initialized to eliminate the problem.

Explicitly initialize all variables (INIT_EIV_5)

Description

jtest! can detect uninitialized variables in your methods.

Example

```
public class INIT_EIV {
    int method () {
        int i;
        return j;
    }
}
```

Diagnosis

Unused variable "i" in method "method."

Repair

These errors can be corrected by initializing the appropriate variables.

Switch statement with bad case (PT_SBC_1)

Description

jtest! can detect where a missing break or return in your code causes control to flow into another case in a switch.

Example

```
public class PT_SBC {
    void method (int i) {
        switch (i) {
            case 1:
                a = 10;
            case 2: case 3:
                a = 20;
            default:
                a = 40;
        }
    }
}
```

Diagnosis

Missing break or return.

Repair

Add the missing break or return.

Avoid assignments within an if condition (PT_ASI_1)

Description

jtest! can detect assignments in if conditional statements.

Example

```
public class PT_ASI {
    void method (boolean val) {
        if (val = true) {
            k += 1;
        }
    }
    private int k = 10;
}
```

Diagnosis

Assigning "val" in an if-statement.

Repair

Do not assign within an if-statement.

Use "equals" instead of "==" (PT_UEI_1)

Description

The "==" operator is used on strings checks if two string objects are two identical objects. If you simply want to check if two strings have the same value, the "equals" method should be used.

Example

```
public class PT_UEI {
    int method (String s1, String s2) {
        if (s1 == s2) {
            return s1.length () + s2.length ();
        }
        return 2 * s1.length ();
    }
}
```

Diagnosis

Comparing two String objects with "==" does a memory location compare, not a characterwise compare.

Repair

Change "==" to "equals".

Avoid for-statements with empty bodies (PT_FEB_1)

Description

For-statements that are immediately followed by a closing statement semicolon can lead to confusion.

Example

```
public class PT_FEB {
    void method () {
        int i;
        for (i = 0; i < 10; ++i) ;
            System.out.println (i);
    }
}
```

Diagnosis

Avoid for-statements with empty body.

Repair

Provide an explicit block statement, using open and closing braces to indicate the scope of the for loop.

Avoid if-statements with empty bodies (PT_IEB_1)

Description

The following code shows three cases of empty bodies within the if-statement. The semicolons prevent the if/else structures from performing anything, regardless of whether the condition is true or false. If there is a body which you intended to be the body of any of these if structures, it would always be executed.

Example

```
public class PT_IEB {
    void method (int i) {
        if (i == 0)
            ;
        if (i == 0)
            ;
        else
            i += 2;

        if (i < 0)
            i *= -1;
        else
            ;
    }
}
```

Diagnosis

Avoid if-statements with empty bodies.

Repair

Consider providing a body in the if-statements or change the logic of the program.

Avoid dangling else statements (PT_ADE_2)

Description

In the following code, it appears that the programmer's intention is to decrement i if i < 5. The compiler always associates an else with the previous if unless instructed by braces to do otherwise. In our example, the else is associated with the second if; therefore, the decrementation takes place if i = 2. To force the structure to execute as originally planned, use braces to indicate to the compiler that the else matches the first if.

Example

```
public class PT_ADE {
    void method () {
        int i = 5;
        if (i < 5)
            if (i < 2)
                i++;
        else
            i--;
    }
}
```

Diagnosis

Avoid dangling else statements. Possible confusion: which "if" does the "else" belong to?

Repair

Include the first if structure between braces. The compiler will know that the second if structure is the sole statement within the first if block and the else matches the right if.

Don't cast primitive datatypes to lower precision (PT_CLP_2)

Description

By casting to lower precision, you truncate the number which will have a different value falling into an unwanted range.

Example

```
public class PT_CLP {
    void method () {
        double d = 4.25;
        int i;

        i = this.square ((int) d);
    }

    int square (int i) {
        return (i * i);
    }
}
```

Diagnosis

Don't cast primitive datatypes to lower precision. Don't cast double "d".

Repair

Either explicitly test the value before performing a cast or avoid it.

Don't use text labels in switch statements (PT_TLS_2)

Description

The example below shows two situations where errors might occur. Omitting a space between the word case and the value being tested in a switch statement and the textlabel "wronglabel" creates two unused labels. This will cause logic errors. When the controlling value is 3, the structure will not perform the desired action. The textlabel "wronglabel" is incorrect. "Switch" handles only "case" labels.

Example

```
public class PT_TLS {
    static int method (int i) {
        switch (i) {
            case 4:
            case3:
                i++;
                break;
            case 25:
            wronglabel:
                break;
        }
        return i;
    }

    public static void main (String args[]) {
        int i = func (3);
        System.out.println (i);
    }
}
```

Diagnosis

Don't use text labels in switch statement. Didn't you want to write "case 3"?

Don't use text labels in switch statement. Don't use Textlabel "wronglabel" inside switch statement.

Repair

Correct case3 label and remove wronglabel.

Avoid using an object to access a class "static" variable or method (PT_AUO_2)

Description

jtest! can detect when you are using an object to access a static method.

Example

```
final class AClass {
    static void getIt () {
    }
}

public class PT_AUO {
    void method () {
        AClass object = new AClass ();
        object.getIt ();
        AClass.getIt ();
    }
}
```

Diagnosis

Avoid using an object to access a class "static" variable or method. Use "AClass.getIt" instead of "object.getIt"

Repair

When dealing with static methods, be careful how you invoke them.

Provide "default" for each switch statement (PT_PDS_3)

Description

jtest! can detect when there is no default label in a switch statement.

Example

```
public class PT_PDS {
    void method (int i) {
        switch (i) {
        case 1:
            a = 10;
            break;
        case 2:
        case 3:
            a = 20;
            return;
        }  // missing default label
    }
}
```

Diagnosis

Provide "default:" label for each switch statement missing "default:" label.

Repair

Add a default statement.

Do not use embedded assignment (PT_NEA_4)

Description

jtest! can detect when embedded assignments are made. Code using embedded assignments becomes cryptic and difficult to read.

Example

```
public class PT_NEA {
    void method () {
        int i = 2;
        int j = 2;
        short r = 4;
        double d = 2;
        double x = 3;
        int k = 3;

        d = (k = i + j) + r;
        d -= (x = i + j) + r;
        d /= (x /= i + j) + r;
    }
}
```

Diagnosis

Do not use embedded assignments.

Do not nest "=" inside of "=".

Do not nest "-=" inside of "-=".

Do not nest "/=" inside of "/=".

Repair

Do not nest assignments; this could lead to confusion.

Don't compare floating point type (PT_DCF_3)

Description

This rule tells you to avoid testing for equality floating point numbers because it might cause logic errors or infinite loops due to unexpected results of the testing. If you do not avoid such testing, at least be careful when doing so.

Example

```
public class PT_DCF {
    int method (double d) {
        if (d == 1) {
            return 1;
        } else if (i != 2) {
            return 2;
        } else return 3;
    }
}
```

Diagnosis

Don't compare floating point types. Don't compare double variable "d" for equality.

Repair

Use the comparison of floating point numbers reasonably.

Avoid using method parameter names that conflict with class member names (PT_MPC_4)

Description

The example below shows the use of the same names "i" and "j" for parameters and an instance variable and a method, respectively. This might create confusion when using the variable or calling the function and eventually lead to a logic error.

Example

```
public class PT_MPC {

    void method (int i, int j) {
    }

    void j () {
    }

    private int i = 3;
}
```

Diagnosis

Avoid using method parameter names that conflict with class member names. Parameter "i" hides member with same name. Parameter "j" hides member with same name.

Repair

Use different names for parameters and class members (variables and methods) to avoid confusion.

Don't confuse + operator for String concatenation (PT_DCP_4)

Description

The code below shows the trap you might fall into when using "+" the wrong way. The output of the following example is "29" not "11" as you expected.

Example

```
public class PT_DCP {
    static void method () {
        System.err.println ("2 + 9 = " + 2 + 9);
    }
    public static void main (String args []) {
        method ();
    }
}
```

Diagnosis

Don't confuse + operator for String concatenation.

Do not use mathematical "+" operations meshed with string concatenation.

Repair

Calculate the result outside the System.out.println statement and print it.

Avoid hiding inherited static member functions (OOP_AHF_1)

Description

jtest! can detect when inherited static methods are hidden by child classes.

Example

```
public class OOP_AHF {
    static void method1 () {}
}

class OOP_AHF_ extends OOP_AHF {
    static void method1 () {}
}
```

Diagnosis

Avoid hiding inherited static methods. Static method "method1" overrides static method in ancestor.

Repair

The solution will depend on the design of the program, but may be as simple as using the inherited method and removing method declared in the child class.

Avoid hiding inherited instance variables (OOP_AHV_2)

Description

jtest! can detect when inherited instance variables are hidden by members declared in child classes.

Example

```
public class OOP_AHV {
    protected long a = 4;
}

public class OOP_AHV_ extends OOP_AHV {
   protected int a = 5;
}
```

Diagnosis

Avoid hiding inherited instance variables. Protected/Public variable "a" hides a variable in ancestor class OOP_AHV.

Repair

The solution will depend on the design of the program, but may be as simple as using the inherited member and removing the member declared in the child class.

Avoid public and package instance variables (OOP_APV_3)

Description

Instance variables are an implementation detail of your class, and it is a good practice to hide such implementation details from the user of your class. Instead, provide the user with methods to access and/or change such variables.

Example

```
public class OOP_APV {
    int k = 10;
}
```

Diagnosis

Use private instance variables. Make package instance variable "k" private and accessible via public methods.

Repair

Declare the variable private and provide access methods to the variable instead.

List all public and package methods/ data first (OOP_LPF_4)

Description

It is a good habit to order the methods and data in a class so that all public and package wide information is presented first, followed by the protected and private information. This way, the user of your class will find the accessible methods directly underneath the class declaration, and will not be bothered with implementation details described by private and protected data/methods.

Example

```
public class OOP_LPF {
    private int method () {
        return i;
    }

    int method2 () {
        return k;
    }

    protected int k = 10;
    private static int i = 0;
}
```

Diagnosis

List all public and package methods/data first. Package method "method2" declared after private methods/data.

Repair

Reorder the list of methods/data in your class.

Implement interfaces non-trivially or abstract (OOP_IIN_5)

Description

When implementing an interface, make sure that each method declared in the interface either gets defined explicitly in your class, or is explicitly declared to be abstract.

Example

```
abstract public class OOP_IIN implements B {
    public void f () {
        int i = 9;
        i = 10;
    }

    public void g () {}
    abstract public void h ();
}

interface B {
    public void f ();
    public void g ();
    public void h ();
}
```

Diagnosis

Implement interfaces nontrivially or abstract. Method "g" implements interface with trivial body.

Repair

Possibly, the programmer forgot to provide a proper body for the offending method, or did not mean to declare this method, in which case it should be declared to be abstract.

Avoid hiding member variables in member functions (MISC_HMV_2)

Description

Variables declared local to a class method can hide instance variables of the same name, effectively blocking access to the instance variable.

Example

```
abstract class MISC_HMV {
    public int method2 () {
        int i = 5;
    }

    private int i = 0;
}
```

Diagnosis

Avoid hiding member variables in methods. Local variable "i" hides member variable of the same name.

Repair

Change the name of the local variable so it becomes unique.

Provide incremental in for-statement or use while statement (MISC_PIF_3)

Description

jtest! can detect when a for-statement does not have an incremental part. This may be an indication that the incremental part is missing or that the code will be clearer if instead of a for-statement, a while-statement is used.

Example

```
public class MISC_PIF {
    void method () {
        int i = 0;
        for (; i < 1000; ) {
            if (i % 2 == 0) {
                i = 2 * i + 1;
            } else {
                i = i/2;
            }
        }
    }
}
```

Diagnosis

Provide incremental in for-statement or use while-statement. Third argument to "for" statement may be missing.

Repair

Check if third argument to the for-statement is missing. If it is missing, either increment the counter within the for structure or change the for-statement to a while-statement.

Provide condition in for-statement (MISC_PCF_3)

Description

The following example shows a for-statement which has an empty second slot. This might cause infinite loops.

Example

```
public class MISC_PCF {
    void method () {
        int i = 0;
        int j = 0;
        for (;; i++) {
            j++;
            break;
        }
    }
}
```

Diagnosis

Provide condition in for-statement. Condition in "for" statement may be missing.

Repair

Consider providing a condition which will be tested before each new pass through the loop.

Each for statement should have a block (MISC_EFB_5)

Description

Each for-statement should have a block.

Example

```
public class MISC_EFB {
    int i = 3;

    void method () {
        for (int i = 0; i < 10; i++)
            System.out.println (i * i);
            System.out.println (i);
    }
}
```

Diagnosis

Each for statement should have a block. Provide a block: "for (...) { statement (s) }"

Repair

Provide a block for each for statement regardless of the number of statements intended to be executed within the block.

Close stream in finally block (OPT_CS_1)

Description

Programs using certain types of resources should release them in order to avoid resource leaks. This can be done through a finally block which is placed after the last catch block of a try statement. Regardless of the outcome of the program, the finally block gets executed.

Example

```
import java.io.*;

public class OPT_CS {
    public static void main (String args[]) {
        OPT_CS opt_cs = new OPT_CS ();
        opt_cs.method ();
    }

    public void method () {
      try {
        FileInputStream fis = new FileInputStream ("OPT_CS.java");
        int count = 0;
        while (fis.read () != -1)
            count++;
        System.out.println (count);
        fis.close ();
      } catch (FileNotFoundException e1) {}
      } catch (IOException e2) {}
    }
}
```

Diagnosis

Close stream in finally block. Close stream in "finally" part of try statement.

Repair

Add a finally block after the last catch.

Use abbreviated assignment operator (OPT_AAS_3)

Description

In order to write programs more rapidly, you should use the abbreviated assignment operator because doing so causes some compilers run faster.

Example

```
public class OPT_AAS {
    void method () {
        int i = 3;
        String s = "fu";

        i = i + 3 - 3;
        s = s + "bar";
    }
}
```

Diagnosis

Use abbreviated assignment operator. Use the abbreviated assignment operator +=.

Repair

Use the abbreviated assignment operator.

Use "String" instead of "StringBuffer" for constant strings (OPT_USC_3)

Description

Dynamically resizable strings are unnecessary for constant strings.

Example

```
public class OPT_USC {
    String method () {
        StringBuffer s = new StringBuffer ("Hello");
        String t = s + "World!";
        return t;
    }
}
```

Diagnosis

Use "String" instead of "StringBuffer" for constant Strings. Use "String" if you don't want to modify a StringBuffer.

Repair

Replace StringBuffer with String if it is certain that the object will not change. This will reduce the overhead and improve performance.

Use "StringBuffer" instead of "String" (OPT_USB_4)

Description

String operations, like concatenation, are done more efficiently for StringBuffer objects than for String objects.

Example

```
public class OPT_USB {
    String method () {
        String s = "Hello";
        String t = s + "World!";
        s += " Venus";
        return t;
    }
}
```

Diagnosis

Use "StringBuffer" instead of "String". Modifying string Object "s": change to StringBuffer.

Repair

If a String is concatenated, try to replace it with a StringBuffer object.

Don't use the negation operator frequently (OPT_DUN_5)

Description

The negation operator decreases the readability of the program, so it is recommended that you do not use it frequently.

Example

```
public class OPT_DUN {
    int method (boolean b) {
        if (!x)
            return 1;
        else
            return 0;
    }
}
```

Diagnosis

Don't use the negation operator frequently.

Repair

Don't use the negation operator.

Enforce name format of classes (CODSTA_NCL_3)

Description

This rule deals with the name format of classes. Most programmers or groups of programmers develop a set of naming rules for classes. jtest! will enforce the name format of the class using a regular expression.

Example

```
public class CODSTA_NCL {
}
```

Diagnosis

Enforce name format of classes.Name of class "CODSTA_NCL" does not match user specified regular expression "^Z[a-zA-Z]"

Repair

Change the name of the class to match the regular expression.

Enforce name format of interfaces (CODSTA_NIF_3)

Description

This rule deals with the name format of interfaces. Most programmers or groups of programmers develop a set of naming rules for interfaces. jtest! will enforce the name format of the interface using a regular expression.

Example

```
public interface CODSTA_NIF {
}
```

Diagnosis

Enforce name format of interface. Name of interface "CODSTA_NIF" does not match user specified regular expression "^Z[a-zA-Z]"

Repair

Change the name of the interface to match the regular expression.

Enforce name format of exceptions (CODSTA_NE_3)

Description

This rule deals with the name format of exceptions. Most programmers or groups of programmers develop a set of naming rules for exceptions. jtest! will enforce the name format of the exception using a regular expression.

Example

```
public class CODSTA_NE extends RuntimeException {
}
```

Diagnosis

Enforce name format of exception. Name of exception "CODSTA_NE" does not match user specified regular expression "^Z[a-zA-Z]"

Repair

Change the name of the exception to match the regular expression.

Enforce name format of methods (CODSTA_NM_3)

Description

This rule deals with the name format of methods. Most programmers or groups of programmers develop a set of naming rules for methods. jtest! will enforce the name format of the method using a regular expression.

Example

```
public class CODSTA_NM {
    method () {
    }
}
```

Diagnosis

Enforce name format of method. Name of method "method" does not match user specified regular expression "^Z[a-zA-Z]"

Repair

Change the name of the method to match the regular expression.

Enforce name format of class variables (CODSTA_NCV_3)

Description

This rule deals with the name format of class variables. Most programmers or groups of programmers develop a set of naming rules for class variables. jtest! will enforce the name format of the variable using a regular expression.

Example

```
public class CODSTA_NCV {
    int variable;
}
```

Diagnosis

Enforce name format of variable. Name of variable "variable" does not match user specified regular expression "^Z[a-zA-Z]"

Repair

Change the name of the variable to match the regular expression.

Enforce name format of class methods (CODSTA_NCM_3)

Description

This rule deals with the name format of class methods. Most programmers or groups of programmers develop a set of naming rules for class methods. jtest! will enforce the name format of the class method using a regular expression.

Example

```
public class CODSTA_NCM {
    static int method () {
    }
}
```

Diagnosis

Enforce name format of class method. Name of static method "method" does not match user specified regular expression "^Z[a-zA-Z]"

Repair

Change the name of the class method to match the regular expression.

Enforce name format of method parameters (CODSTA_NMP_3)

Description

This rule deals with the name format of method parameters. Most programmers or groups of programmers develop a set of naming rules for method parameters. jtest! will enforce the name format of the method parameter using a regular expression.

Example

```
public class CODSTA_NMP {
    void method (int i) {
        if (i == 10) return;
    }
}
```

Diagnosis

Enforce name format of method parameter. Name of method parameter "i" does not match user specified regular expression "^Z[a-zA-Z]"

Repair

Change the name of the method parameter to match the regular expression.

Enforce name format of instance variables (CODSTA_NIV_3)

Description

This rule deals with the name format of instance variables. Most programmers or groups of programmers develop a set of naming rules for instance variables. jtest! will enforce the name format of the instance variable using a regular expression.

Example

```
public class CODSTA_NIV {
    private int i = 0;
}
```

Diagnosis

Enforce name format of instance variable. Name of instance variable "i" does not match user specified regular expression "^Z[a-zA-Z]"

Repair

Change the name of the instance variable to match the regular expression.

Enforce name format of local variables (CODSTA_NLV_3)

Description

This rule deals with the name format of local variables. Most programmers or groups of programmers develop a set of naming rules for local variables. jtest! will enforce the name format of the local variable using a regular expression.

Example

```
public class CODSTA_NLV {
    void method () {
        int i = 0;
        i++;
    }
}
```

Diagnosis

Enforce name format of local variable. Name of local variable "i" does not match user specified regular expression "^Z[a-zA-Z]"

Repair

Change the name of the local variable to match the regular expression.

Order group members with the same name together (CODSTA_OGM_3)

Description

This rule enforces various standards to improve readability.

Example

```
public class CODSTA_OGM {
    void foo () {
    }

    void bar () {
    }

    void foo (int a) {
    }
}
```

Diagnosis

Order group members with same name together.

Repair

Move the members called "foo" together.

Use uppercase letters for final static variables (CODSTA_USV_3)

Description

Use uppercase letters for final static variables.

Example

```
public class CODSTA_USV {
    final int i = 3;
}
```

Diagnosis

Use uppercase letters for final static variables. Change final static variable "i" to uppercase for readability.

Repair

Change final static variable "i" to uppercase for readability.

Put the main function last (CODSTA_PML_4)

Description

This rule makes the program comply with various coding standards regarding the form of class definitions.

Example

```
public class CODSTA_PML {
    public static void main (String args[]) {
    }
    void foo () {
    }
}
```

Diagnosis

Put main method last.

List method main () last within the class definition.

Repair

List the "main" method last within the class definition.

Never throw Class Exception directly (CODSTA_NTX_4)

Description

jtest! can detect when Exceptions are being used too generally.

Example

```
public class CODSTA_NTX {
    void method () throws Exception, ArithmeticException {
        throw new Exception ();
    }

    void foo () throws Exception {
        Exception e = new Exception ("TEST");
        throw e;
    }
}
```

Diagnosis

Never throw Class "Exception" directly. Only throw subclasses of "Exception."

Repair

Concentrate on handling subclasses of Exception.

Never use Exception or RuntimeException in catch (CODSTA_NCE_4)

Description

jtest! detects when you are trying to catch Exception or RuntimeException. They are too general.

Example

```
public class CODSTA_NCE {
    void method () {
        try {
        } catch (Exception e1) {
        }
        try {
        } catch (RuntimeException e2) {
        }
        try {
        } catch (Throwable e3) {
        }
    }
}
```

Diagnosis

Never use Exception or RuntimeException in catch.

Don't catch "Exception" directly.

Don't catch "RuntimeException" directly.

Don’t catch “Throwable” directly

Repair

Deal with subclasses of Exception, RuntimeException, and Throwable when handling exception.

Never throw an "Error" directly (CODSTA_NTE_4)

Description

Throwing an "Error" directly is too general.

Example

```
public class CODSTA_NTE {
    void method () {
        throw new LinkageError ();
    }

    void method2 () {
        throw new Error ();
    }
}
```

Diagnosis

Never throw an "Error" directly.

Never throw a subclass of "Error" directly.

Repair

Explicitly deal with the subclasses of a superclass derived from Error to avoid potential bugs.

Use conventional variable names (CODSTA_CVN_5)

Description

Use conventional variable names for one-character names.

b for a byte
c for a char
d for a double
f for a float
i, j, and k for integers
l for a long
o for an Object
s for a String
v for an arbitrary value of some type

Example

```
public class CODSTA_CVM {
    void method () {
        int b = 1;
        int d = 1;
    }
}
```

Diagnosis

Use conventional variable names.

Use variable "b" only for a byte.

Use variable "d" only for a double.

Repair

Use conventional variable names.

Bad tag in javadoc comment (JAVADOC_BT_2)

Description

jtest! will notify you if there is a bad tag in the javadoc comments of your code. A doc-comment may use special tags which all begin with the @ character and allow javadoc to provide additional formatting for the documentation. Check the HTML markup tags which the javadoc comments may contain.

Example

```
/**
 * This is a comment
 * @unsupported tag
 */

public class JAVADOC_BT {

}
```

Diagnosis

Bad tag in javadoc comment. Unsupported javadoc tag.

Repair

Use only supported tags.

Distinguish between javadoc and ordinary comments (JAVADOC_DC_5)

Description

jtest! can detect javadoc comments. If the javadoc comment in your program does not end with **/, jtest will point that out to you.

Example

```
/**
 *  This is a comment
 */

public class JAVADOC_DC {
}
```

Diagnosis

Distinguish between ordinary comments and javadoc comments.

Repair

Add an extra star to the end of the comment to distinguish it from ordinary comments.

jtest! Tutorial

Welcome to the jtest! Tutorial. This tutorial begins by describing and walking you through the four types of testing that jtest! performs (static analysis, white-box testing, black-box testing, and regression testing), then describes how you might apply various jtest! techniques when testing one of your own directories, zip files, or jar files.

Please note that although the four types of tests are discussed separately, jtest! can perform static analysis, white-box testing, regression testing (on all test runs after the first), and black-box testing (on classes whose test case outcomes you have validated) with just one click of the **Start** button.

This tutorial contains the following lessons:

- Lesson 1 - Performing White-Box Testing
- Lesson 2 - Performing Static Analysis
- Lesson 3 - Performing Black-Box Testing
- Lesson 4 - Performing Regression Testing
- Lesson 5 - Testing a Set of Classes

Lesson 1 - Performing White-Box Testing

White-box testing checks that the class is structurally sound. It doesn't test that the class behaves according to the specification, but instead ensures that the class doesn't crash and that it behaves correctly when passed unexpected input.

White-box involves looking at the class code and trying to find out if there are any possible class usages that will make the class crash (in Java this is equivalent to throwing an uncaught runtime exception).

jtest! completely automates white-box testing. jtest! executes the class under test using a symbolic virtual machine (a virtual machine that uses mathematical equations instead of numbers to internally represent the variable states). While executing each bytecode, it checks if there are solutions to the equations that will make the bytecode throw an uncaught runtime exception. If it finds a solution it reports an error for it. The equations are also solved to find test cases that execute as many branches in the code as possible.

How to perform white-box testing - Overview

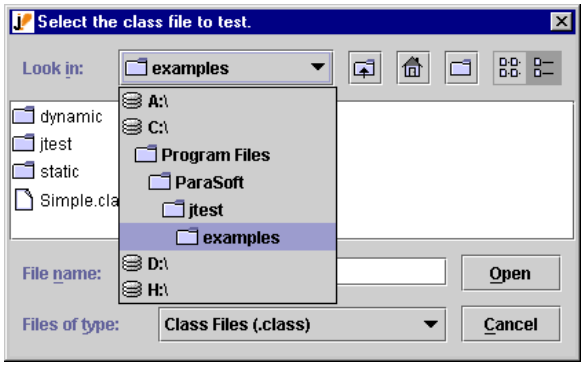
By default, jtest! is configured to perform white-box testing, so all you need to do is tell jtest! what class or set of classes to test and click the **Start** button.

If you test a project, results will be displayed in the **Class Name> Errors Found> Uncaught Runtime Exceptions** branch of the Project UI's Results panel.

If you test a single class, results will be displayed in the **Uncaught Runtime Exceptions** branch of the Errors Found panel.

How to perform white-box testing - Example

1. Open jtest!
2. Browse to Simple.class in the jtest! examples directory using the **Browse** button in the Class Name panel.

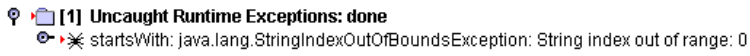


3. Click the **Start** button in the toolbar.

jtest! will perform static and dynamic analysis on the class. A dialog box will open to notify you when testing is complete. Information on test progress will be displayed in the Test Progress panel. Problems found will be reported in the Errors Found panel.

Once testing is completed, you will see two errors reported in the Errors Found panel. For this example, we're going to focus on the uncaught runtime exception found.

The Errors Found panel will list the following uncaught runtime exception:


[1] Uncaught Runtime Exceptions: done
startsWith: java.lang.StringIndexOutOfBoundsException: String index out of range: 0

This error message reveals that there is some input for which the class will throw an uncaught runtime exception at run-time. This could cause the application running this class to crash.

To see a stack trace like the one the Java virtual machine gives when an uncaught runtime exception is thrown, expand this branch.

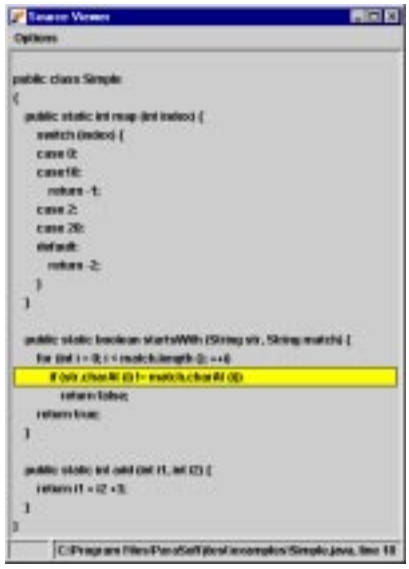
To see an example usage of this class that leads to the reported uncaught runtime exception, expand the **Test Case Input** branch.

[1] Uncaught Runtime Exceptions: done
startsWith: java.lang.StringIndexOutOfBoundsException: String index out of range: 0
| at java.lang.String.charAt (0)
| at Simple.startsWith ("", "0") [Simple.java, line 19]
Test Case Input
| boolean RETVAL = Simple.startsWith ("", "0");

The error reveals that the "startsWith" method is implemented incorrectly. The method should return false for the argument "" and "0" instead of throwing a runtime exception.

If the error is not fixed, any application using this class will eventually crash or give incorrect results when used.

To view the source code of the class with the problematic line of the stack trace highlighted, double-click the node containing the exception's file/line information.



Viewing Automatically Generated Test Cases

To see all the test cases that jtest! automatically generated for this test, click the **View** button in the toolbar. The View Test Cases window will open.

In the View Test Cases window, expand the **Automatic Test Cases** branch You will see a list of this class's methods. Click any method with a number greater than zero by it, and open the **Test Cases** branch, or press **Control** and click your right mouse button, then choose **Expand Children** from the shortcut menu that opens. All the inputs that jtest! generated for each method are now displayed.

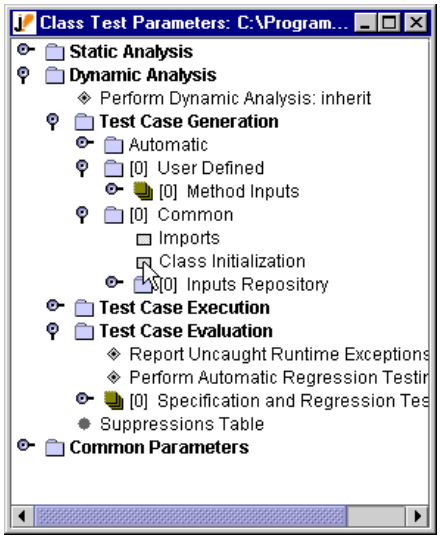
Setting an Object To a Certain State

In some cases, you may want to set up an initial state prior to testing a class. For example, suppose that a class is used as a global object containing static member variables accessible by any other project within the application. When jtest! tests an object that uses this static member variable, a NullPointer Exception will result because the variable has not been set. This problem can be solved by giving jtest! initialization code.

For example, the file ExecGlobal.java contains a static variable that is assigned by its ancestor Exec, which is defined in the Exec.java file. In this example, the instantiation on the Exec object is performed by the"Exec object's "main" method, which is defined in the Exec.java file. When jtest! tests the Test object's method doTest, it does not know that the condition of a valid instance of Exec is available in the ExecGlobal object. Thus, class initialization needs to be performed.

To perform this initialization:

1. Make sure all the java files involved have been compiled.
2. Open the Class Test Parameters for the class you want initialization code to be called for.
3. In the Class Test Parameters window, open **Dynamic Analysis> Test Case Generation> Common**.
4. Double-click the **Class Initialization** node.

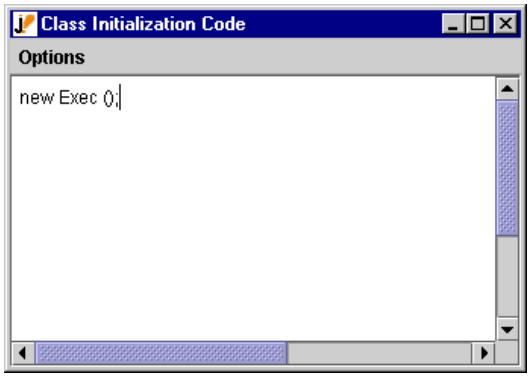


The Class Initialization Code window will open.

5. Enter the following initialization code in the Class Initialization Code window.

```
new Exec ();
```

This will create an instance of the "Exec" object before the class "Test" is tested.



6. (Optional) If you want to check the method, choose **Save & Check** from the **Options** menu.
7. From the **Options** menu, choose **Save**.
8. From the **Options** menu, choose **Quit**.

Lesson 2 - Performing Static Analysis

Static analysis can be thought of as consisting of both an automatic code review and automatic coding standards enforcement.

Code reviews have been found to be very useful in uncovering errors and logic flaws in code. In a code review, someone other than the original programmer looks for possible errors and problems in code that already compiles.

Adhering to coding standards is an effective error prevention strategy. Coding standards are rules that ensure that code is written in a way that makes it less error-prone. By adhering to these standards, you significantly reduce the opportunities to introduce errors in the code. Adhering to coding standards enforcement also makes code more readable and easier to maintain.

When performing static analysis, jtest! statically analyzes the classes by parsing the .java source and applying a set of rules to it, then alerting you to any rule violations in the class(es) under test.

How to perform static analysis- Overview

By default, jtest! is configured to perform static analysis, so all you need to do is tell jtest! what class or set of classes to test and click the **Start** button.

If you test a project, results will be displayed in the **Class Name> Errors Found> Static Analysis** branch of the Project UI's Results panel.

If you test a single class, results will be displayed in the **Static Analysis** branch of the Errors Found panel.

How to perform static analysis - Example

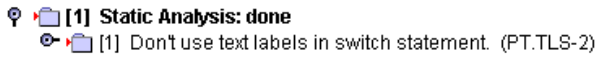
1. Open jtest!.
2. In the Class Testing UI, use the **Browse** button in the Class Name panel to choose Simple.class (in the jtest! examples directory).
3. Click the **Start** button in the toolbar.

jtest! will perform static and dynamic analysis on the class.

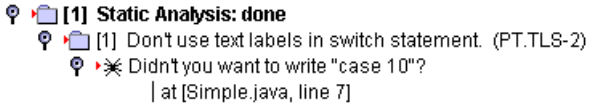
Information on test progress will be displayed in the Test Progress panel. Errors found will be reported in the Errors Found panel.

Once testing is completed, you will see two errors reported in the Errors Found panel. For this example, we're going to focus on the Static Analysis error.

The **Static Analysis** branch of the Errors Found panel should list the following error:



Fully expand the error message so you can see more details about the violation.

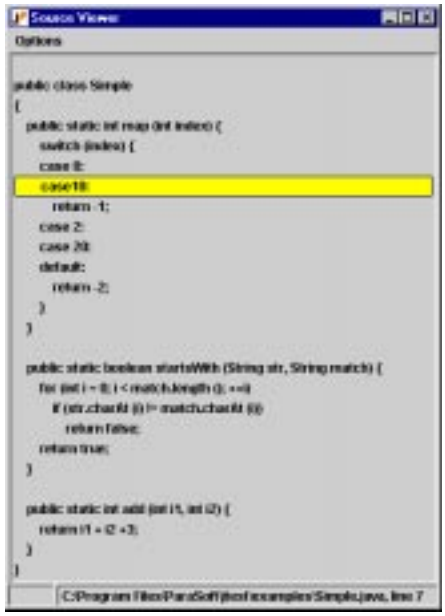

[1] Static Analysis: done
[1] Don't use text labels in switch statement. (PT.TLS-2)
Didn't you want to write "case 10"?
at [Simple.java, line 7]

This error reveals that the developer inadvertently wrote "case10" instead of "case 10". If the class is not fixed, it will give incorrect results when it is passed the value 10.

To learn more about any rule that jtest! enforces. right-click the topmost branch of the error violation you would like explained, then choose **View Rule Description** from the shortcut menu.

A dialog box that contains a description and example of the selected rule violation, as well as a suggested repair, will open.

You can also view the source code responsible for the violation; to do this, simply double-click the error message you want to see the source code for, then view the code in the source code viewer that opens.



Customizing Static Analysis

You can change the rules that jtest! enforces by enabling or disabling specific rules, or entire rule severity categories.

Note: A description of each rule, as well as an example violation and suggested repair, appears in the Rules section of the Reference Guide.

Enabling/Disabling Specific Rules

1. View which rules are currently enabled/disabled.
    - Before setting up a test, open the Global Test Parameters window by clicking the **Global** button.
    - In the Global Test Parameters window, open the **Static Analysis** folder.
    - Open the **Rules** node.
    - Open the **Built-in Rules** node.
    - Open the node category whose rules you would like to view.
2. Enable/disable rules by right-clicking the appropriate rule and choosing either **Enable** or **Disable** from the shortcut menu.

Enabling/Disabling Rule Severity Categories

Each rule has a severity category associated with it (This severity is indicated by the final character in each rule code. For example, UC_AUV-2 has a severity level of 2, and INIT_EIV-5 has a severity level of 5.) There are five severity levels; one is the most severe and five is the least severe. Only severity levels one, two, and three are enabled by default.

jtest! looks for violations to a rule only if the rule is enabled *and* its severity level is also enabled.

Thus, you may also want to configure which severity categories are enabled and disabled. To change this in the Global Test Parameters, open the **Severity Level** folder (beneath **Static Analysis> Rules)**, and enable/disable categories by right-clicking a category whose status you want to change and choosing **Set to true** or **Set to false** from the shortcut menu.

By providing you with an easy, flexible way to enforce Java coding standards, jtest! helps you do what some consider the most essential task in ensuring software quality: prevent errors. Preventing as many errors as possible from entering the code translates not only to less time, effort, and money spent finding and fixing errors as the project progresses, but also to a significantly reduced risks of having errors that elude testing and make their way to the end-user.

Lesson 3 - Performing Black-Box Testing

Black-box testing checks that the class behaves according to the specification, meaning that the class produces the correct output (outcomes) for each input.

In general, any black-box testing tool requires the user to specify a set of inputs to try, along with the correct outputs for those inputs. The tool then runs the inputs automatically and checks that the correct outputs are generated.

Black-box testing becomes much simpler when you use jtest!

jtest! automatically generates a set of sophisticated inputs; jtest! analyzes the class bytecodes and provides a minimal set of inputs that produce as much coverage as possible for the class.

In addition to using these automatically generated inputs, you can also specify an additional set of inputs in the standard fashion: by listing, for each class method and argument, the inputs that jtest! should try.

When you start the test, jtest! executes all the inputs and provides you with the outcomes for those inputs, organized in a simple tree representation. You can then see the outcomes and verify them with one click of the mouse, so when jtest! performs subsequent tests on this class, it will be able to notify you when specification and regression testing errors occur.

Note: To perform black-box testing with jtest, you must have the JDK compiler on your PATH. The 'javac' command is the compiler and it is usually located under the bin directory of most JDK installations.

How to perform black-box testing

By default, jtest! is configured to perform black-box testing as long as the outcomes from the inputs from automatically generated and/or user-defined test cases have been validated.

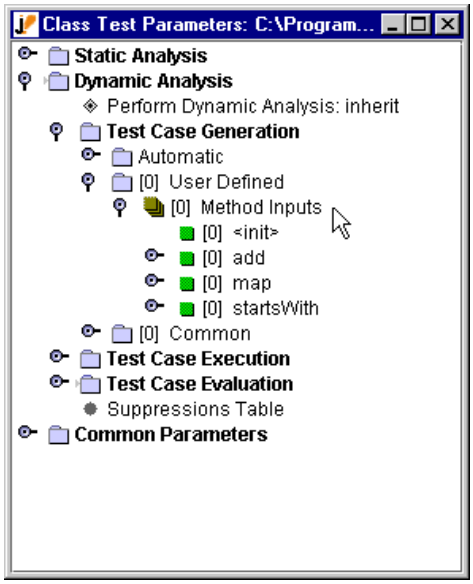
The following steps explain how to add user-defined inputs and how to validate the outcomes for these inputs. Once the inputs are defined and the file outcomes have been validated, jtest! will perform black-box testing automatically each time you test a class or set of classes.

This example uses the Class Testing UI, but mentions (*in italics*) steps that need to performed differently if this test is performed from the Project Testing UI.

Adding Inputs to Static Methods

To add user-defined test cases:

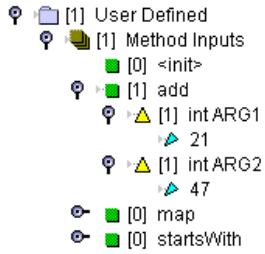
1. Copy Simple_ok.java to Simple.java and compile Simple.java (both classes are located in the jtest! examples directory).
2. Open jtest! and browse to Simple.class in the Class Name panel.
   Note: If you were performing black-box testing via the Project Testing UI, you would perform the following steps instead of the above step:
   - *Open jtest!*
   - *Click the Project button to open the Project Testing UI*
   - *In the Search In field, browser to the directory, zip file, or tar file that contained this class.*
3. Click the **Params** button in the Class Testing UI to open the Class Test Parameters window.
   Note: If you were performing black-box testing via the Project Testing UI, you would open the Class Testing Parameters via the Project Testing Parameters window.
4. Open **Dynamic Analysis**.
5. Open **Test Case Generation**.
6. Open **User Defined**.
7. Open **Method Inputs**.



Now that a list of all methods in the class is displayed, you can specify inputs for the method arguments. For example to specify that for the method "add" jtest! should also test it with arguments 21 and 47; you can enter these inputs by performing the following steps:

1. Open **add**.
2. Right-click **int ARG1**, then choose **Add Input Value** from the shortcut menu.

3. Type "21" then press **Enter**.
4. Right-click **int ARG2**, then choose **Add Input Value** from the shortcut menu.
5. Type "47" then press **Enter**.



Now run the test again:

1. Close the Class Test Parameters window.
2. Click the **Start** button to test the class.

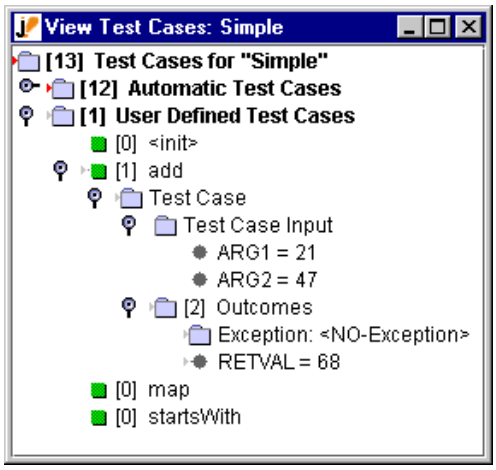
Validating outcomes

To see and validate the user-defined inputs and outcomes:

1. Click the **View** button in the toolbar open the View Test Cases window.

Note: If you were testing in the Project Testing UI, you would open the View Test Cases window by right-clicking the node associated with this class's name in the lower Results window, then choosing View Test Cases from the shortcut menu.

2. Open **User Defined Test Cases**.
3. Open **add**.
4. Fully expand **Test Case**.



The View Test Cases window displays all user-defined and automatically generated test cases, along with their outcomes.

To indicate that the outcomes are correct, right-click the **Outcomes** node and choose **Set All Correct** from the shortcut menu.

Note: If all outcomes were not correct, you could indicate whether or not the outcome for each test case was correct by right-clicking the appropriate outcome, and choosing **Correct** (if the listed outcome was the expected outcome), **Incorrect** (if the listed outcome was not the expected outcome), **Unknown** (if you didn't know how the listed outcome compared to the expected outcome), or **Ignore** (if you wanted jtest! to ignore the listed outcome) from the shortcut menu.

Now every time jtest! is run on that class, jtest! will run that user-defined test case and will check that the outcomes are correct.

Viewing Automatically Generated Test Cases

You can also see the inputs that jtest! generated automatically by fully expanding the **Automatic** branch in the View Test Cases window. Because jtest! automatically generates a set of inputs, you will normally want to run the class once in the completely automatic mode of testing (simply opening a class and clicking **Start**) before you add your own inputs. Then you can decide if you want to test the class with additional inputs, and, if so, add them as user-defined inputs.

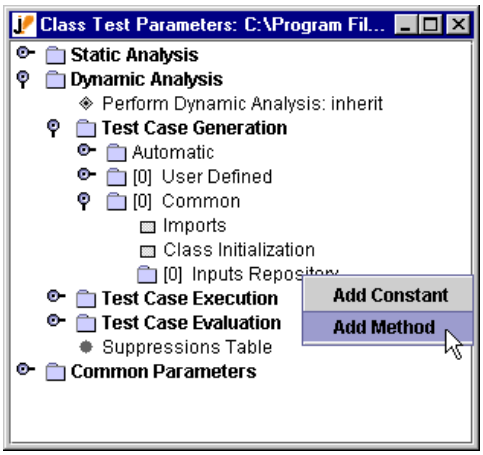
By automatically providing a core set of sophisticated inputs, jtest! reduces the number of test cases that you need to create; this allows you to test a greater portion of the program than you could if all inputs were designed manually, in less time.

Note that you can also validate the outcomes from the automatic test cases; if you do this, jtest! will perform black-box testing for both the automatic test cases and the user-defined test cases.

Adding Object Type Inputs to Instance Methods

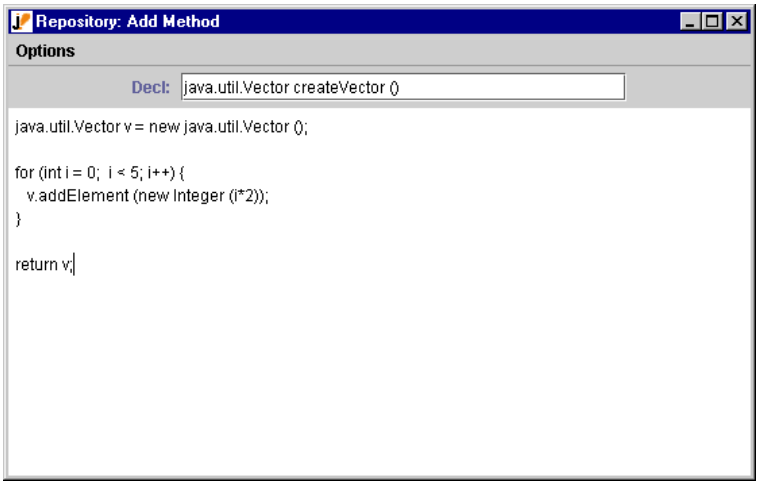
To see how to add inputs to instance methods, perform the following steps using the example code provided:

1. Compile Test.java.
2. In the Class Testing UI, click the **Browse** button and open the Test.class file that you just produced.
3. Click the **Params** button. The Class Test Parameters window will open.
4. In the Class Test Parameters window, open **Dynamic Analysis**.
5. Open **Test Case Generation**.
6. Open **Common**.



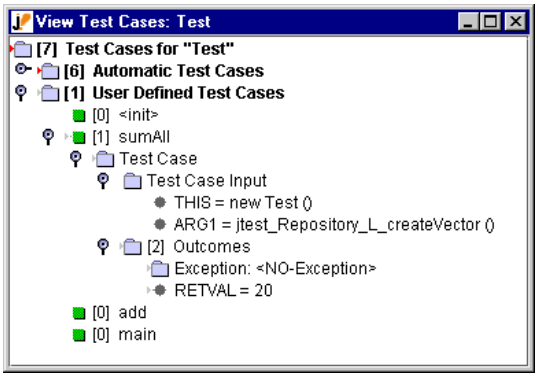
7. Right-click **Inputs Repository** then choose **Add Method** from the short-cut menu.
   The Add Method window will open.
8. In the Add Method window, add the method declaration.
   In the **Decl** field of the Add Method window, type:
   ```
   java.util.Vector createVector ()
   ```
9. Add the method body.
   In the main pane of the Add Method window, type:

```
java.util.Vector v = new java.util.Vector ();

for (int i = 0; i < 5; i++) {
    v.addElement (new Integer (i*2));
}

return v;
```



10. From the **Options** menu, choose **Save** to save this method.
11. From the **Options** menu, choose **Quit** to exit this window.
12. Add the implicit 'this' argument.
    - In the Class Test Parameters windows, go to **Dynamic Analysis> Test Case Generation> User Defined> Method Inputs> sumAll> Test THIS**.
    - Right-click **Test THIS** then choose **Add Input Value** from the shortcut menu. A text-field appears below **Test THIS**.
    - In the text field, type
      `new Test ()`
      then press **Enter**.
13. Add the input for sumAll's ARG1 from the Inputs Repository.
    - In the Class Test Parameters Windows, go to **Dynamic Analysis> Test Case Generation> User Defined> Method Inputs> sumAll> java.util.Vector ARG1**.
    - Right-click **java.util.Vector ARG1** then choose **Add From Local Repository> CreateVector ()** from the shortcut menu.

14. Run the test by clicking the **Start** button.
15. View the results.
    - Click the **View** button to open the View Test Cases window.
    - In the View Test Cases window, open **User Defined Test Cases**.
    - To see the input and outcome of the test case you defined, fully expand the **sumAll** branch.



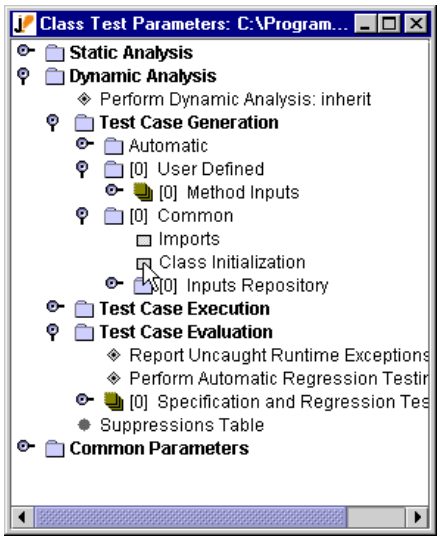
Setting an Object To a Certain State

In some cases, you may want to set up an initial state prior to testing a class. For example, suppose that a class is used as a global object containing static member variables accessible by any other project within the application. When jtest! tests an object that uses this static member variable, a NullPointer Exception will result because the variable has not been set. This problem can be solved by giving jtest! initialization code.

For example, the file ExecGlobal.java contains a static variable that is assigned by its ancestor Exec, which is defined in the Exec.java file. In this example, the instantiation on the Exec object is performed by the"Exec object's "main" method, which is defined in the Exec.java file. When jtest! tests the Test object's method doTest, it does not know that the condition of a valid instance of Exec is available in the ExecGlobal object. Thus, class initialization needs to be performed.

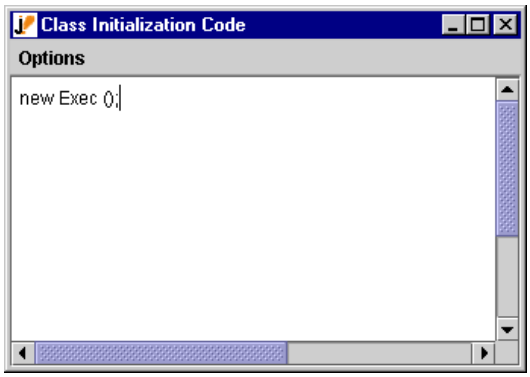
To perform this initialization:

1. Make sure all the java files involved have been compiled.
2. Open the Class Test Parameters for the class you want initialization code to be called for.
3. In the Class Test Parameters window, open **Dynamic Analysis> Test Case Generation> Common**.
4. Double-click the **Class Initialization** node.



The Class Initialization Code window will open.

5. Enter the following initialization code in the Class Initialization Code window.
   `new Exec ();`
   This will create an instance of the "Exec" object before the class "Test" is tested.



6. (Optional) If you want to check the method, choose **Save & Check** from the **Options** menu.
7. From the **Options** menu, choose **Save**.
8. From the **Options** menu, choose **Quit**.

Comparing Outcomes for Methods That Return Object Types

jtest! checks Object values by representing them using the “toString” method and some public instance methods chosen heuristically. If the methods jtest! uses to determine (and compare) the state of the return value of a method when the return value is of Object type are not enough, or do not represent state as you like, you can use a method called jtestInspector () which jtest! will use to represent the Object state.

For example, if the return value of your method is “MyObject”, you would want to define a ‘jtestInspector’ in MyObject and that ‘jtestInspector’ would return a representation of the state of the MyObject in the format of a string.

Inspector.class demonstrates an example usage of jtestInspector. To see how jtestInspector can be used:

1. Test Inspector.class
    - Open Inspector.class in the Class Testing UI.
    - Click the **Start** button.
2. View the first test case generated for the method “make.”
    - Click **View** to open the View Test Cases window.
    - Fully expand **Automatic Test Cases> “make”> Test Case 1**. If you look at the **Outcomes** branch, you will see that the Object return value “RETVAL” is represented using the ‘jtestInspector’ methods.

Lesson 4 - Performing Regression Testing

Regression testing checks that the class behavior doesn't change. Regression testing gives you the peace of mind of knowing that the class doesn't break when the code is modified.

Regression testing is very similar to black-box testing in that both check whether certain inputs generate certain outputs. Whereas black-box testing is performed to verify that the class performs according to specification, regression testing aims to assure that the class *continues* to perform according to specification after it has been modified.

jtest! provides automatic regression testing. Even if you don't specify what the correct outcomes are, jtest! remembers the outcomes from previous runs and compares them every time the class is tested and reports an error for any outcome that changes.

How to perform regression testing - Overview

All you need to do to perform automatic regression testing is:

1. Restore previously saved test parameters by choosing **File> Open**, then typing or selecting the test parameters you want to restore.
   Note: You can also open recently used parameters by choosing **File> Open Recent> [File Name]**.
2. Run the test by clicking the **Start** button.

If you tested a project, results are displayed in the **Class Name> Errors Found> Specification and Regression Errors** branch of the Project UI's Results panel.

If you tested a single class, results are displayed in the **Specification and Regression Errors** branch of the Errors Found panel.

How to perform regression testing - Example

jtest! doesn't display any regression errors on the first run through a class because it is impossible for jtest! to give a regression error the first time it runs a class. Regression testing checks that the class outcomes don't change, so it always needs a first run for reference. Consequently the first thing you need to

do in this example is run Simple.class once, to provide jtest! with some reference values.

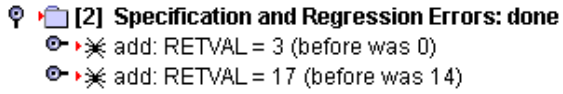
1. Start the jtest! Class Testing UI.
2. Browse to Simple.class in the jtest! examples directory using the **Browse** button in the Class Name panel.

3. Click the **Start** button in the toolbar.
4. Wait for the jtest! to complete this test.

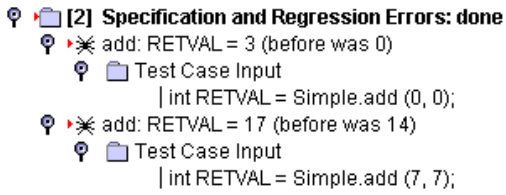
To see how regression testing works, we will introduce an error into Simple.java and run it again

1. Copy Simple_re.java into Simple.java (both classes are located in the jtest! examples directory).
2. Recompile Simple.java.
3. Retest Simple.java.

Now, along with the errors you saw before, jtest! reports the following regression errors:



Expand the error messages, then the **Test Case Input** branches to see the inputs for which that regression errors occur. The first error tells us that the code has changed and that the method "add" is now returning 3 instead of 0 for the input 0, 0. The second error reveals that the method "add" is now returning 17 instead of 14 for the input 7,7.


[2] Specification and Regression Errors: done
add: RETVAL = 3 (before was 0)
Test Case Input
int RETVAL = Simple.add (0, 0);
add: RETVAL = 17 (before was 14)
Test Case Input
int RETVAL = Simple.add (7, 7);

By automating regression testing, jtest! makes keeping your classes error-free easier than ever.

Lesson 5 - Testing a Set of Classes

At this point, you've seen how the main types of testing can be performed using the Class Testing UI. Now, we will describe how you might apply various testing techniques while testing one of your own projects using the Project Testing UI.

Starting the Test

Before you start a test, you need to open the Class Testing UI. You can open this UI by clicking the **Project** button in the Class Testing UI.

To start a test, open the directory, jar file, or zip file you want to test by clicking the **Browse** button and choosing the project that you want to test.

Fully Automatic Testing

You might want to start your testing of this directory by performing static analysis and white-box testing on all of your classes; this testing requires almost no user intervention, and is a critical step in preventing and detecting crash-causing errors. To perform these two types of tests, simply click the **Start** button.

Errors found will be reported in the **[Class Name]> Errors Found** branch of the Results panel in the Project Testing UI.

Static Analysis Results

If you want to focus on the static analysis results, you can view only the results for classes that have static analysis errors by right-clicking the **[Class Name]> Errors Found> Static Analysis** node in the upper results window, then selecting **Show Results for this Category** from the shortcut menu.

Note: To see results for all classes in the lower Results window, right-click the **All Classes** node in the upper Results panel, then choose **Show Results for this Category** from the shortcut menu.

To get more information about a particular error, go to the lower Results panel and fully expand the **Class Name> Errors Found> Static Analysis> [error message]** node.

To view a description of the rule that was violated, right-click the static analysis error message, then choose **View Rule Description** from the shortcut menu. To

view the source with the line containing the violation highlighted, double-click the node that contains the file/line information for the violation.

To enable or disable a rule or rule severity category, click the **Global** button and modify static analysis parameters in the Global Test Parameters window.

White-Box Testing Results

To focus on the uncaught runtime exceptions found, go to upper Results window, right-click the **[Class Name]> Errors Found> Uncaught Runtime Exceptions** node, and choose **Show Results from this Category** from the shortcut menu. To get detailed information about an error, go to the lower Results window and expand any node. The exception's stack trace is contained in the **[Class Name]> Errors Found> Uncaught Runtime Exceptions> [error message]** node, and the calling sequence is contained in the **[Class Name]> Errors Found> Uncaught Runtime Exceptions> [error message]> Test Case Input** node.

If a reported uncaught runtime exception is actually the correct behavior of the class, or if you want jtest! to ignore the outcome of an input while checking for specification and regression errors, right-click the error message, then choose the appropriate command (either **Not an Error** or **Ignore this Outcome**) from the shortcut menu.

To see the test cases that jtest! automatically generated for a class, right-click the **[Class Name]** node in the lower Results window, and choose **View Test Cases** from the shortcut menu. This will open the View Test Cases window that contains all test case inputs and outcomes.

Coverage Information

To see the coverage this initial test achieved, open the **[Class Name]> Test Progress> Total Coverage** node.

Focusing on a Single Class's Results

If you want to focus on a single class's results, you can open the class in the Class Testing UI; if a class is open in this UI, you will see only the errors related to the selected class. To open a class's results in the Class Testing UI, right-click the **[Class Name]** node in the lower Results window, then choose **Load in Class Testing UI** from the shortcut menu.

Black-Box Testing

Now you might want to perform black-box testing on a few of the classes in the project. Black-box testing involves three main steps:

1. Adding inputs.
2. Running the test.
3. Validating outputs.

Adding Inputs

When you are testing a project, you add inputs for each class individually, via the Class Test Parameters window. One way to edit class test parameters from the Project UI is listed by performing the following steps:

1. In the Project Testing UI, click **Params** to open the Project Test Parameters. The Project Test Parameters window will open.
2. Open the **Classes in Project** branch.
3. Right-click the node that corresponds to the class whose parameters you want to modify. A shortcut menu will open.
4. Choose **Edit Class Test Parameters** from the shortcut menu. The Class Test Parameters window associated with that class will open.

Note: When a class has been tested in the Project Testing UI, you can open its class test parameters by right-clicking the [Class Name] node, then choosing **Edit Class Test Parameters** from the shortcut menu.

Once you are in the Class Test Parameters window, you can add inputs to static and instance methods as well as set the state of objects. Procedures for performing these actions are described in Lesson 3.

Running the test

To run the test, simply click the **Start** button. jtest! will then perform all the tests it performed in previous test runs, plus it will perform white-box testing for the user-defined test cases.

Validating Outcomes

You can see and evaluate each class's test case inputs and outcomes in the View Test Cases window. To open this window from the Project Testing UI, right-click the appropriate [Class Name] node, then choose **View Test Cases** from the shortcut menu. At this point, you can validate outcomes as described in Lesson 3.

The next time you run jtest!, all available tests (white-box testing, static analysis, black-box testing, regression testing) will be performed. If any specification or

regression errors occur, they will be reported in **[Class Name]> Errors Found> Specification and Regression Errors**.

Regression Testing

After you modify one or more of the classes in your project, you will probably want to check whether or not your changes introduced any problems. You can gather such information by performing regression testing. To perform regression testing, simply restore the project test parameter by selecting **Open** from the Project Testing UI's **File** menu, then clicking the **Start** button. Any specification and regression testing errors that occur will be reported in **[Class Name]> Errors Found> Specification and Regression Errors**.

Index

A

arrow colors 58

B

batch mode 64
black-box testing
- about 40
- adding constants and methods 46
- adding inputs 45
- adding object type inputs to instance methods
  - tutorial 202
- adding primitive inputs 45
- performing in Class Testing UI 41
- performing in Project Testing UI 43
- setting an object's state
  - tutorial 190, 206
- tutorial 197

buttons
- Class Testing UI 79
- Find Class UI 128
- Project Testing UI 89

C

calling sequence
- viewing in Class Testing UI 15
- viewing in Project Testing UI 26

class
- opening in Class Testing UI (project test) 28

class initialization code 37
- tutorial 190, 206

Class Name panel 82
class test 6
- example 8, 41
- performing 6
- results 13, 15
- saving/restoring test and test parameters 51
- tutorial 187, 193, 197, 209

class test parameters 108
- common 117
- dynamic analysis 110

editing (project test) 29
static analysis 109
Class Testing UI 76
Class Name panel 82
Errors Found Panel 85
menus 77
Test Progress Panel 83
toolbar 79
code reviews 30
coding standards 30
coding standards enforced 130
command-line jtest! 64
compiling a source 54
constants, adding 45
coverage
viewing in Class Testing UI 15, 83
viewing in Project Testing UI 24, 26
cursors 75

D

detailed project report 61
directory, testing. See project test
dynamic analysis
about 34
customizing 36
example 8
performing 35
performing only 7

E

error detection 1
error prevention 1, 30
Errors Found Panel 13, 85
example source
viewing in Class Testing UI 15
viewing in Project Testing UI 26

F

Filter-in 18
Find Classes UI 128

G

Global History 52
global test parameters 95
common 104

dynamic analysis 99, 121
static analysis 96
GUI. See UI

H

help, context sensitive 53
history for test 52

I

initial state, setting 37
tutorial 190, 206
input that caused error
viewing in Class Testing UI 15
viewing in Project Testing UI 26
inputs
adding 45
adding constants and methods 46
adding primitive inputs 45
restricted 111
inputs repository 45
tutorial 202
inputs, adding 45
tutorial 197, 214
installation 2
instance methods
tutorial 202

J

jar file, testing. See project test
jtest!
introduction 1
quick start 4
jtestInspector 56, 57, 116, 208

L

license 2

M

menus
Class Testing UI 77
Project Testing UI 87
methods, adding 46

N

Number of Errors Found window 22

O

object type inputs
    tutorial 202
object types 208
outcomes
    validating 41, 58
        tutorial 214
    viewing 55
    viewing reference outcomes 14

P

parameters
    inheritance 65
    search 125
    test 65
ParaSoft, contacting 3
progress
    viewing in Class Testing UI 83
    viewing in Project Testing UI 23
Project Controls Panel 93
project report 60
project test 17
    history 52
    performing 17
    results 22, 26
    saving/restoring test and test parameters 51
    tutorial 212
project test parameters 119
    common, search, classes in project 124
    dynamic analysis 121
    static analysis 120
Project Testing UI
    menus 87
    Project Controls Panel 93
    Results Panel 94
    toolbar 89

Q

Quality Consulting 3

R

regression testing 49
    example 11
    performing 50

tutorial 209, 215
report 60
detailed project 61
project 60
single class 60
summary project 61
restoring test parameters 51
restricted inputs 111
results
class test 13, 15
project test 22, 26
removing from Project Testing UI 27
results directory 62
Results For All Classes window 23
Results Panel 94
Results panel 22
rules 130
enabling/disabling severity categories 32
enabling/disabling specific rules 32
list of built-in rules 130
viewing rule descriptions in Class Testing UI 15
viewing rule descriptions in Project Testing UI 26

S

saving test parameters 51
search parameters 125
single class report 60
Skip List 18
source
compiling 54
editing 54
viewing 54
source location, indicating 106
specification testing, about 40
stack trace
viewing in Class Testing UI 15, 26
static analysis 30
customizing 32
example 8
list of built-in rules 130
performing 31
performing only 7
results 31
suppressions 67

tutorial 193, 212
summary project report 61
suppressions
dynamic analysis 69
from Class Testing UI 16
from Project Testing UI 27
static analysis 67
viewing reason for suppression 57

T

technical support 3
test case
adding 45
automatic generation 38
evaluation
modifying from Class Testing UI 16
modifying from Project Testing UI 27
reference test cases 115
validation 55
viewing test cases 55
test history 52
Test Only List 18
test parameters 65
class 108
global 95
project 119
saving and restoring 51
Test Progress Panel 83
Testing 20
"THIS" object 111
toolbar
Class Testing UI 79
Project Testing UI 89
trees 74
tutorial 186
black-box testing 197
project test 212
regression testing 209, 215
static analysis 193, 212
white-box testing 187, 212

U

UI
Class Testing UI 73, 76

configuring startup UI 73
Find Classes UI 128
Project Testing UI 73

V

validating outcomes 41, 55, 58
tutorial 214
View Test Cases window 55

W

white-box testing 38
example 8
performing 39
tutorial 187, 212

Z

zip file, testing. See project test.